AD A090095

(2) LEVEL II

NAVAL POSTGRADUATE SCHOOL

Monterey, California

DTIC
ELECTE
S OCT 10 1980 D
B

(9) Master's THESIS

(6) COMPARISON AND ANALYSIS
OF SOME ALGORITHMS FOR IMPLEMENTING
PRIORITY QUEUES

by

(10) Alinur Goksel (12) 138

(11) Jun[redacted]80

Thesis Advisor: D. R. SMITH

Approved for public release: distribution unlimited

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A090 095	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Comparison and Analysis of some Algorithms for Implementing Priority Queues		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis: June 1980
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Alinur Goksel		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1980
		13. NUMBER OF PAGES 137
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release: distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Priority queue, Analysis of Algorithms, Worst Case Analysis, Heap, K-ary tree, Single Linked-list, Leftist tree, Linked tree, AVL-tree, 2-3 tree, Fixed Priority

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A priority queue is a data structure for maintaining a collection of items, each having an associated key, such that the item with the largest key is easily accessible. Priority queues are implemented by using heap, k-ary tree, single linked-list, leftist tree, linked tree, AVL tree, 2-3 tree and fixed priority property. Required storage for each method was obtained and the worst case

Cont.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
(Page 1) S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

time analysis was done in terms of key comparisons and key exchanges during the insertion and deletion process.

Finally, each of these methods were run on PDP-11 UNIX TIME SHARING SYSTEM at NPS using different random number generators to get the average CPU time for insertion and deletion process.

Accession For	
NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	23

Approved for public release; distribution unlimited.

Comparison and Analysis
of some Algorithms for Implementing
Priority Queues

by

Alinur Goksel
Lieutenant (Junior Grade), Turkish Navy
Turkish Naval Academy, 1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

June 1983

Author --

Approved by: --
Thesis Advisor

Second Reader

Chairman, Department of Computer Science

Dean of Information and Policy Sciences

ABSTRACT

A priority queue is a data structure for maintaining a collection of items, each having an associated key, such that the item with the largest key is easily accessible. Priority queues are implemented by using heap, k-ary tree, single linked-list, leftist tree, linked tree, AVL-tree, 2-3 tree, and fixed priority property. Required storage for each method was obtained and the worst case time analysis was done in terms of the number of key comparisons and key exchanges during the insertion and deletion process.

Finally, each of these methods were run on PDP-11 UNIX TIME SHARING SYSTEM at NPS using different random number generators to get the average CPU time for insertion and deletion process.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

The author wants to express his grateful thanks to Assistant Professor Douglas R. Smith for his guidance, assistance and suggestions during the time of this study.

I. INTRODUCTION AND BACKGROUND

Priority queues are implemented by using heap, k-ary tree, singly linked list, leftist tree, linked tree, AVL tree, 2-3 tree and fixed priority property in this research. In order to understand how these implemented methods work, a reader needs to be familiar with the following concepts.

A data type is a term which refers to the kinds of data that variables may "hold" in a program. The simplest data types are the fixed-point values, floating-point numbers, logical values, and characters all represented as sequences of bits. These elementary data types are considered to be primitive data structures. However, to provide for processing of more complex information, it is necessary to construct more complicated data structures to serve as the internal representation of the information.

Data object is a term referring to a set of elements, say D. For example the data object integer refers to D={0,±1,±2,...}. D may be finite or infinite and if D is very large, special ways of representing its elements in a computer may be needed.

A data structure can be defined as, the description of the set of objects and, the specification of the set of operations which may legally be applied to elements of the data object. For integers, there would be the arithmetic operations +,-,*,/ and perhaps many others such as mod, div, greater than, less than, etc. The data object integer plus a

description of how +,-,*,/, etc. behave constitutes a data structure definition [14]. Data structures are designed to meet two basic requirements:

i) To represent the external information in a unique and unambiguous fashion.

ii) To facilate efficient manipulation of the data by the computer[10].

A node(usually called a record or a structure) is a collection of data and links. Each item in a node is called a field. A field can contain an array or a primitive data items, such as a character string, integer value, floating point value, or it can contain a pointer to another node[10]. A tree is a graph (set of nodes connected by edges) in which there are no loops and has a root(i.e., a node from which every other node can be reached by following the edges in their proper directions).

Trees are a species of nonlinear structure of considerable importance in computer science, partly because they provide natural representations for many sorts of nested data that occur in computer applications, and partly because they are useful in solving a wide variety of algorithmic problems. There are several varieties of trees which can be defined in a representation-independent manner. The particular one which will be more interested in this research is binary trees:

A binary tree is a finite set of elements, called nodes, which is empty or else satisfies the following:

i) There is a distinguished node called the root, and

ii) the remaining nodes are divided into two disjoint subsets, L and R, each of which is a binary tree. L is the left subtree of the root and R is the right subtree of the root[12].

Binary trees are represented on paper by diagrams such as the one in figure 1. If a node w is the root of the left(right) subtree of a node v, w is the left(right) child of v and v is the parent or father of w. A <u>leaf</u> in a tree is a node which has no descendants. Leaves can also be called terminal nodes or external nodes, while nonleaves will be called internal nodes. The <u>degree</u> of a node in a tree is the number of subtrees it has. The <u>level</u> of the root is 0 and level of any other node is one plus the level of its parent. The <u>height</u>(sometimes called the depth) of a tree is the maximum of the levels of its leaves. A <u>complete binary tree</u> is a binary tree with leaves on at most two adjacent levels d-1 and d in which the leaves at the bottommost level d lie in the leftmost positions of d [13]. The second binary tree in figure 1 is complete binary tree. <u>Balanced binary tree</u> is a binary tree with leaves on at bottommost two levels.

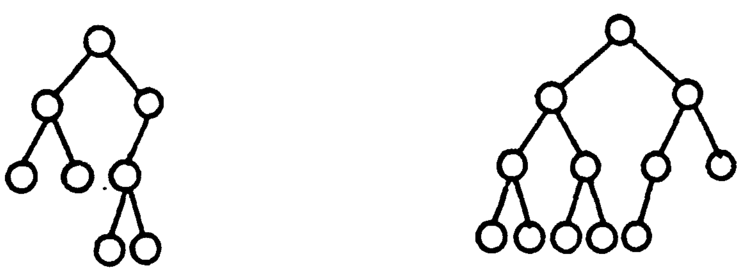


Figure 1.

Stacks and queues are sequences of items, which are permitted to grow and contract only by following special disciplines for adding and removing items at their end points.

In a stack, all insertions and deletions are done at only one end of a sequence. Stacks have the property that the last item inserted is the first item that can be removed and for this reason, they are sometimes called LIFO lists, after this "last-in,first-out" property.

In a queue, all insertions are done at one end, called the rear or back, and all deletions are done at the other end, called front. Queues have the property that the first item inserted is the first item that can be removed and, for this reason, they are sometimes called FIFO lists, after this "first-in,first-out" property. Queues implicitly provide a linear order for their items corresponding to their "order of arrival". Thus, queues are used where we wish to process items under a "first-come,first-served" discipline[13].

In addition to LIFO and FIFO disciplines there is a "largest-in,first-out" discipline. A data structure which follows this discipline is called a priority queue.

A. WHAT IS A PRIORITY QUEUE

A priority queue is a data structure which holds data items with an associated priority. Items can be inserted in the priority queue in an arbitrary order but at any given time only the item with the highest priority in the priority queue may be accessed or deleted. More precisely, if Q is a priority queue and X is an item containing a priority from a linearly-ordered set, then following operations are defined:

Create()..... Create an empty priority queue.

Ismtpq(Q).... Test whether a priority queue is empty.

Insert(X,Q).. Add item X to the collection of items in Q.

Delete(Q).... Remove the item with the highest priority on the Q. If Q is empty then delete Q.

Best(Q)...... Return the item with the highest priority

Pfunc(X)..... Compute the priority of the item X.

Trees are mostly used to implement priority queues. As it will be seen in the next chapters, node representations are assumed to contain only a priority field, pointers to another nodes, and some integer fields. But, in fact, in addition to this information, there may be other pieces of information or pointers to particular events or jobs which have to be executed. A node structure in a priority queue can be in the form:

KEY	I	auxilliary information

where the KEY field contains the priority of the item, and

field 'I' contains either data, events or jobs which have to be executed. In the calculation of a storage requirement for each method in the following section, 'I' will be used to indicate the size of either data,events or jobs. Auxilliary information field contains pointers to another nodes, counts, balance factor,... etc.,which provide to maintain the priority queue structure.

B. PRIORITY QUEUE APPLICATIONS

Priority queues can be used in sorting and selection problems. The idea of selection sorting is to fill the priority queue using successive 'insert' operations, and then emptying the queue by using 'delete' operations repeatedly[3]. An algorithm for this application of a priority queue has been given below where Q is the priority queue.

Input: An array A of N items.

Output: Array A sorted into non increasing order.

```
BEGIN
  FOR i=1 to N DO
   insert(Q, A[i]); (*add A[i] to the collection*)
                    (* of items in Q. *)
  FOR i=1 to N DO
   A[i]:=delete(Q); (* put the item with highest priority*)
                    (* into output area. *)
END.
```

Priority queues also arise in certain numerical iterations. One scheme for adaptive quadrature maintains a priority queue of subinterval whose union constitutes the interval of integration; each subinterval is labeled with the estimated error committed in the numerical integration over it. In each step of the iteration, the subinterval with the largest error is removed from the queue and bisected.

Then the numerical integration is performed over these two smaller subintervals, which are inserted into the queue. The iteration stops when the total estimated error is reduced below a prescribed tolerance[3].

An obvious application of priority queues is in operating and industrial practice for the scheduling of jobs according to fixed priorities. In this situation jobs with priorities attached enter a system, and the job of highest priority is always the next to be executed. But, in order to prevent a low-priority job from being delayed indefinitely, the restriction to fixed priorities may be violated[3].

Priority queues also improve the efficiency of some well-known graph algorithms. In Kruskal's algorithm for computing minumum spanning trees(a minumum spanning tree is a network of n nodes connected by edges with least cost, where the cost of a network is the sum of distances of its edges), the procedure of sorting all edges and then scanning through the sorted list can be replaced by inserting all edges into a priority queue and then successively deleting the smallest edge[2,4]. Similar applications have been found for priority queues in shortest path problems which commonly arise on networks [4,5,6]. An algorithm in a pascal-like language for computing spanning tree by using a priority queue has been given below, where G is the set of edges in the minumum spanning tree and Q is a priority queue.

Input: Set of edges each having a cost value associated with it.

Output: Set of edges which satisfy minumum spanning tree property.

```
BEGIN
    G=[]; (* G is initially empty. *)
    FOR i=1 to N DO
     insert(Q,edge i); (* add edge i to the collection *)
                       (* of edges in Q. *)
    FOR i=1 to N DO
     edge=delete(Q);(* remove the edge with minumum cost.*)
     IF edge can be added to G without forming a loop THEN
       G=G + edge; (*add edge to the collection of edges
                   (* in G. *)
END.
```

Priority queues can also be used in the implementations of branch-and-bound algorithm. In particular they are used to implement the 'best-first' strategy (also known as branching from the largest upper bound) [16].

Finally, Charters' prime number generator uses a priority queue in a scheme to reduce its internal storage requirements[4,8]. B. L. Fox has mentioned that priority queues are useful in implementing some discrete programming algorithms[3,4].

II. IMPLEMENTATION AND THE WORST CASE ANALYSIS

In this section of the research, priority queue is implemented by using a property of Heap, K-ary tree, Single linked-list, Leftist tree, Linked tree, AVL-tree, 2-3 tree, and Fix priority. A node structure and brief algorithm for each method has been given.

An addition to these methods indicated above, a priority queue scheme can also be implemented by using a P-tree structure which was discovered in 1964 by Arne Jonassen and Ole-Johan Dahl and a binomial queue structure which was discovered in 1975 by Jean Vuillemin [3]. An implementation and detail analysis of a binomial queue structure has been studied by Brown Mark Robbin in his Ph. D. Thesis at Stanford University.

The analysis of algorithms is quite important in computer programming, because there are usually several algorithms available for a particular application and we would like to know which is best.

An implemented method in this research requires various amounts of storage and time to perform it. In this section, the amount of spaces required to store N items in a computer has been determined for each method. The worst case efficiency of a priority queue scheme is defined in terms of the number of inter-key comparisons and exchanges during an insertion and deletion process. This analysis is done in the following way:

Generally, a method which uses a binary tree structure take $\lfloor \log_2 N \rfloor$ time to execute. Because, they are usually implemented as a recursive procedures, and they call itself recursively at most as many times as the height of a tree which is equal to $\lfloor \log_2 N \rfloor$ if a tree is balanced.

LEMMA: The height of a complete binary tree with N nodes is equal to $\lfloor \log_2 N \rfloor$.

PROOF: A complete binary tree of height h has at most 2^h external nodes. Therefore, the number of internal nodes, n, is bounded above by $n \le 2^h - 1$ (since the number of internal nodes is a sum of the form $1+2+4+\ldots+2^{h-1}$, which is a geometric series with sum $2^h - 1$). The total number of nodes, N, is bounded by $2^h \le N \le 2^{h+1} - 1$. From this,

$$h \le \log_2 N \le \log_2 (2^{h+1} - 1) < h + 1$$

thus,

$$h = \lfloor \log_2 N \rfloor .$$

Since, each execution of a recursive procedure requires a fix number of inter-key comparisons(C), and exchanges(E), the efficiency of an algorithm would be equal to $C\lfloor \log_2 N \rfloor$ times inter-key comparisons and $E\lfloor \log_2 N \rfloor$ times inter-key exchanges in the worst case.

A. HEAP

DEFINITION: A heap is a binary tree with some special properties. A tree is a heap if and only if it satisfies the following conditions.

1) All internal nodes(with one possible exception) have degree 2, and at level d-1 (where 'd' is the depth of the tree) the leaves are all to the right of the internal nodes. The right most internal at level d-1 may have degree 1 (with no right son).
2) The priority at any node is greater than or equal to the priority at each of its sons (if it has any son)[12].

IMPLEMENTATION: Heaps are not difficult to implement. A data structure array is used to implement the heaps. If A is an array, and n is the size of the array then the locations of A are numbered from 1 to n. This numbering operation is done automatically by the computer when the array is created. The integer value 'i' is used as an index to refer to the i'th location of the array. (where 1 <= i <= n)

These locations of the array can be represented as a node in the binary tree. This numbering is done from left to right and level by level (beginning with the root), and illustrated in figure 2.

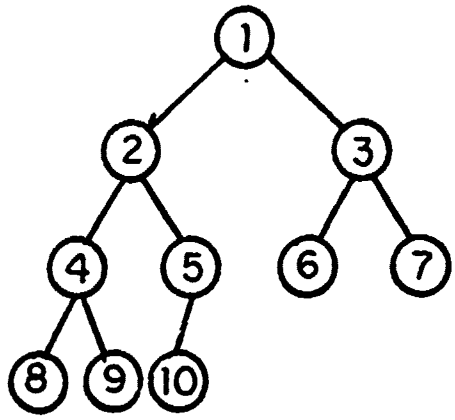


Figure 2

If the array A is used to implement the heap, the second condition in the definition of the heap can be formalized as follows ;

$A[i] >= A[2i]$, and

$A[i] >= A[2i+1]$ for $(1 <= i <= j/2)$

where j is the number of items in the heap.
Figure 3(a) illustrates the binary tree representation of the heap with 10 items in it. (b) illustrates the array representation of the same heap.

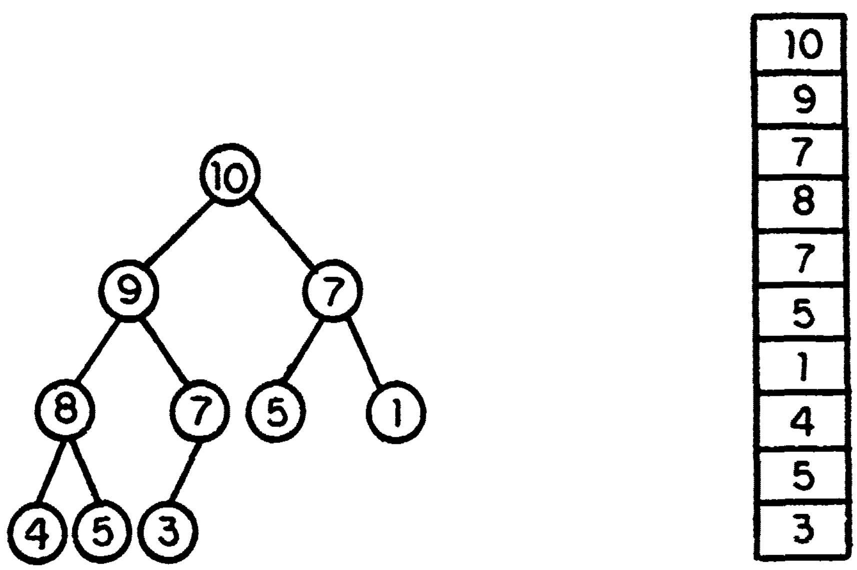


Figure 3

INSERTION: Let's suppose an array A is already created and index i is set to 1 to indicate the current last position of the heap. To insert the new item, first put it into current last position in the array and call SIFTUP to satisfy heap property. The algorithm for the siftup process is given below. Figure 4 illustrates the insertion process.

```
PROCEDURE SIFTUP(i)
/* Adjust the binary tree with the last position 'i' to
satisfy the heap property */
   IF i <= 1 THEN exit
   ELSE
     IF A[i] is greater than its father THEN
       BEGIN
        exchange A[i] and its father.
        siftup(father's position of i)
       END
END SIFTUP.
```

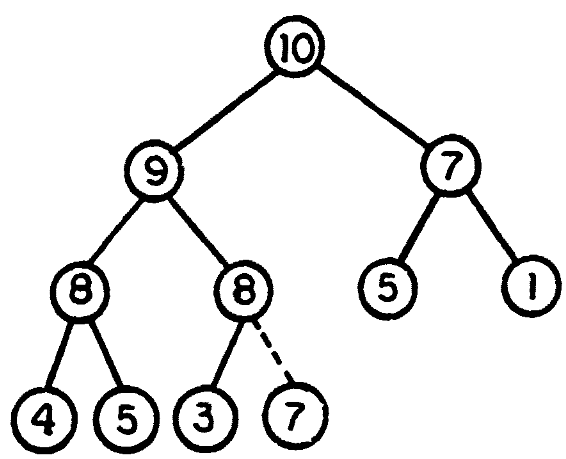


Figure 4. After insertion of priority 8 into figure 3(a).

DELETION: The highest priority item is always at the root.Save it in another place and put the last item in the first position. Decrease the number of items in the heap by 1, and call procedure siftdown to satisfy the heap property. The algorithm for siftdown is given below. Figure 5 illustrates the deletion process.

```
PROCEDURE SIFTDOWN(i,k)
/* Adjust the binary tree with root i and the last position
k to satisfy the heap property. The left and right subtrees
of root 'i',i.e., with roots 2i and 2i+1 already satisfy the
heap property. */
    IF i is a leaf node THEN exit
    ELSE
      Find largest son of 'i'
      IF the largest son is greater than A[i] THEN
       BEGIN
          exchange the largest son and A[i]
          siftdown(largest son's position,k)
       END
END SIFTDOWN.
```

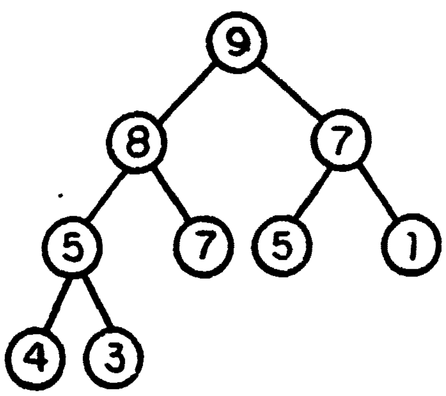


Figure 5. After deletion of highest priority from figure 3(a).

A heap property is usually used in heapsort algorithm which sorts one sequence of length n. A heapsort algorithm is slightly different from a heap algorithm which is used to implement a priority queues in this research. Let A be an array initially containing keys K_1, K_2, K_3 .....K_n in locations A[1], A[2],.....A[n]. An algorithm heapsort which is given below takes the array A as an input and sorts its keys into nondecreasing order. Heapsort algorithm takes O(nlog n) steps to sort all sequences of length n [1].

```
PROCEDURE HEAPSORT
/* This is the complete heapsort algorithm which takes an
array of elements A[i], 1 <= i <= n as an input and arranges
the elements of A sorted into nondecreasing order.*/
BEGIN
 buildheap;
 FOR i=n STEP -1 UNTIL 2 DO
  interchange A[1] and A[i]
  siftdown(1,i-1)
END HEAPSORT.
```

```
PROCEDURE BUILDHEAP
/* This procedure takes an array of elements A[i], 1 <= i <=
n and gives all of A the heap property. */
 FOR i=!n/2! STEP -1 UNTIL 1 DO
  siftdown(i,n)
END BUILDHEAP.
```

HEAP INSERTION WORST CASE ANALYSIS: Suppose a binary tree with N-1 items in it which already satisfy the heap property. In the insertion process the worst case occurs if the priority of the new item is bigger than the priority of the root. The number of key exchanges and the key comparison required to find the proper position for the newly inserted item would be equal to the height of the tree. Because, at each level there would be one key comparison and one key exchanges between children and its father. In the worst case, the new item would come to rest in a root, and the height of the tree would be equal to $\lfloor \log_2 N \rfloor$ after insertion. These comparisons and exchanges are handled by procedure siftup and procedure siftup calls itself recursively at most $\lfloor \log_2 N \rfloor$ times.

HEAP DELETION WORST CASE ANALYSIS: If the height of the tree is equal to $d = \lfloor \log_2 N \rfloor$ after the deletion of the highest item, the number of key comparisons which would be done by procedure siftdown, will be equal to 2d. This is because the rightmost item at the bottommost level had been moved to the root and has to be sifted down to satisfy the heap property after deletion. At each run of the procedure siftdown, there would be two key comparisons; one between left and right son to find the bigger son and one between the bigger son and its father. The worst case occurs if the item at the root comes to rest in a leaf at the bottommost level, which in

this case procedure siftdown calls itself recursivly at most d times.

The number of key exchanges would be equal to d+1: because the item at the root could be exchanged with bigger child of it along any downward path at most d times before coming to rest, and one exchange has been done between the first and last items before the siftdown process.

STORAGE REQUIREMENT FOR A HEAP: A heap uses an array to hold a priority of an item. A potential drawback of heaps is that they require a sufficiently large block of contiguous storage to be allocated in advance; because the size of the queue at any given time is not known and an array as big as the maximum size of the queue has to be allocated. At any given time, if there are N items in the queue, required total storage would be equal to N units for priority fields and N*I units space for the information where, I is the size of information at each node.

B. K-ARY TREE

DEFINITION: A k-ary trees are a generalization of binary trees. A k-ary tree is a finite set of elements called nodes, which is empty or else satisfies the following:

i) There is a distinguished node called the root, and

ii) the remaining nodes are divided into k disjoint subsets, each of which is a k-ary tree [12].

The priority at any node is greater than or equal to the priority at each of its sons (if it has any).

IMPLEMENTATION: Implementation of the k-ary trees is very similar to the implementation of the heaps. An array is also used to implement nodes in the k-ary tree. The numbering of nodes in the 4-ary tree is illustrated in figure 8.

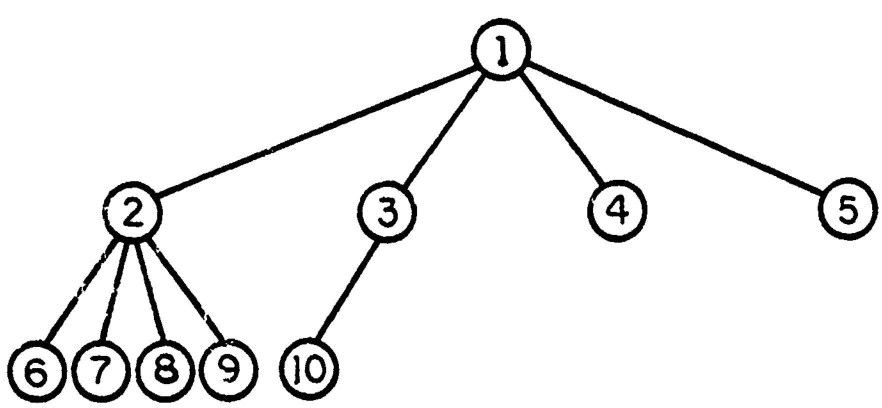


Figure 8

If A is an array and there are n items in it. location A[n+1] contains zero as a terminal symbol to indicate the

successor position of the last item in the k-ary tree. This terminate symbol is used in deletion process.

Figure 7(a) illustrates the complete 4-ary tree with 10 items in it. 7(b) illustrates array representation of the 4-ary tree.

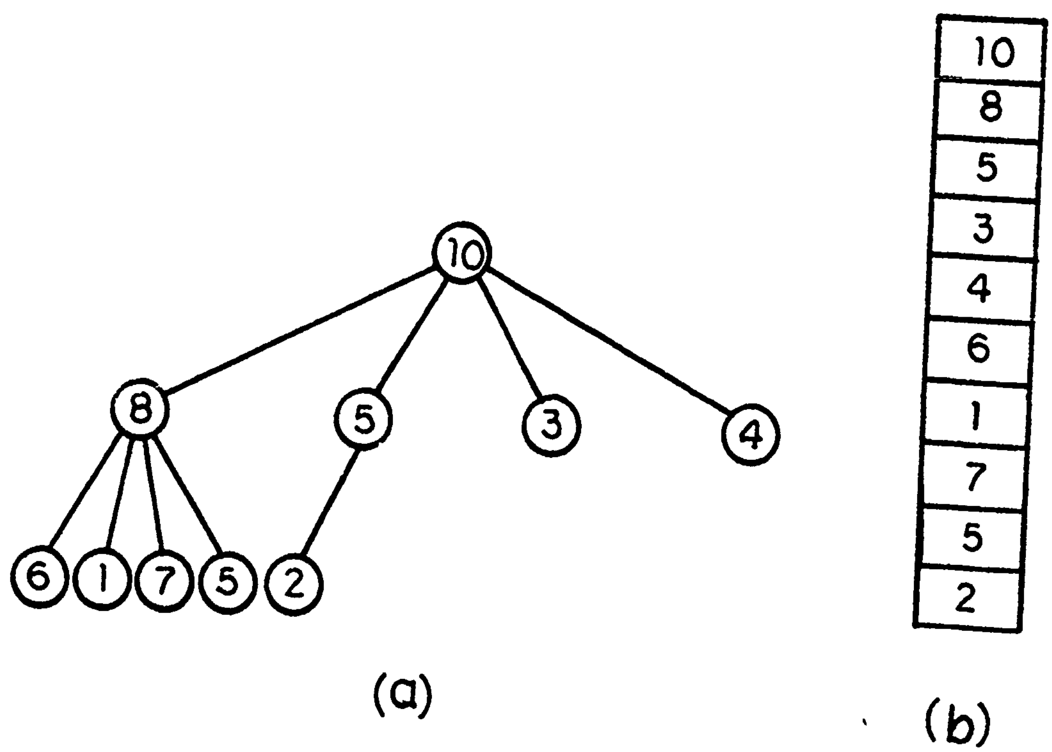


Figure 7

Since the number of nodes on the successive levels of a complete k-ary tree follows a geometric progression 1, k, k^2, k^3,...., the relations shown below can be used to compute the parents, and the children, with the proviso that, for a node to exist, its node number must lie in the range 1 to N, where N is the total number of nodes in the tree.

LEMMA:

1) parent(n) = (n + k -2) div k

2) ith child of(n) = k(n -1) + i + 1 (for 1<=i<=k)

PROOF:

for 2-ary tree:

i) parent(n)=(n + 2 -2) div 2 = n div 2

ii) left child of(n)= 2(n-1) + 1 + 1 = 2n

iii) right child of(n)=2(n-1) + 2 + 1 = 2n + 1

The relation (ii) can be proved by induction on n;
For n=1, clearly the left child is at 2 unless 2 > N in which case 1 has no left child.
Now assume that for all j, 1 <= j < n, left child(j) is at 2j.
Then, the two nodes immediately preceding left child of (n) in the representation are the right child of (n-1), and the left child of (n-1). The left child of n can be obtained by adding 2 to the left child of (n-1) ;

$$2(n-1) + 2 = 2n - 2 + 2 = 2n$$

unless 2n > N in which case n has no left child.

Relation (iii) is an immediate consequence of (ii) and the numbering of nodes on the same level from left to right. Relation (i) follows from (ii) and (iii). This is true because of a characteristic of the 'div' operation on integers. I.e., parent(2n)=2n div 2=n and (2n+1) div 2=n, where operation n div k can be defined as the floor of n/k, denoted $\lfloor n/k \rfloor$, stands for the greatest integer less than or equal to n/k.

For k-ary tree:

i) parent(n)=(n+k-2) div k

ii) first child of (n)=k(n-1) + 1 + 1=k(n-1) + 2

iii) i'th child of (n)=k(n-1) + i + 1 (for 1 <= i <= k)

The relation (ii) can be proved by induction on n, by following the same method as above.

For n=1, clearly the left child is at 2 (k(1-1)+2=2) unless 2 > N in which case 1 has no left child.

Now assume that for all j, 1 <= j < n, the first child of (j) is at k(j-1)+2.

Then, the k nodes immediately preceding the first child of (n) in the representation are the k'th child of (n-1), the (k-1)th child of (n-1),.....,the second child of (n-1), and the first child of (n-1). The first child of (n) can be obtained by adding k to the first child of (n-1) ;

k((n-1)-1) + 2 + k = k((n-1)-1+1) +2 =k(n-1) + 2

unless k(n-1) + 2 > N in which case n has no first child.

The relation (iii) is an immediate consequence of (ii) and the numbering of nodes on the same level from left to right. Relation (i) follows from (ii) and (iii). (This is true because of a characteristic of the 'div' operation on integers).

I.e., parent(k(n-1)+i+1) = (k(n-1)+i+1+k) div k

= (kn+i-1) div k   1<=i<=k

for the first child (i=1) --> kn div k = n

.
.
.

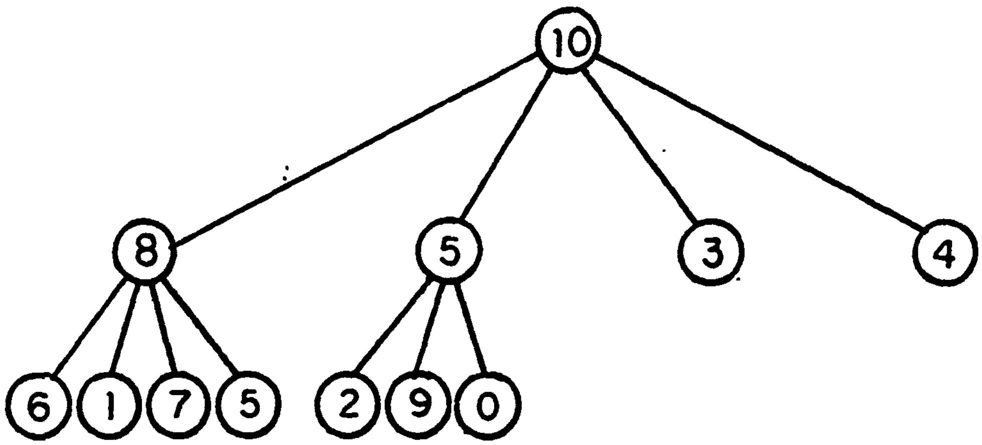
for the k'th child (i=k) --> kn+k-1 div k = n.

INSERTION: Let's assume the array A is already created and i is set to 1 to indicate current last position in the array. In order to insert the new item, first put it in the current last position of the array, and terminate symbol zero in its successor position, and call procedure SIFTUP to siftup the newly inserted component A(n) into its proper position. Figure 8 illustrates insertion process into figure 7.

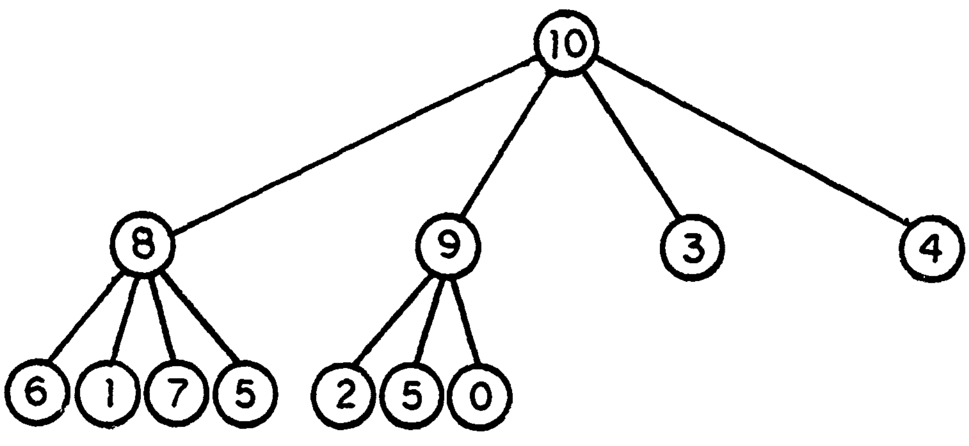
```
PROCEDURE SIFTUP(i)
/* start with item A(i), find its father and compare them.
   If it is necessary, exchance them to satisfy k-ary
   property. Process will be stoped when it is reached
   to the root. */
IF i <= 1 THEN  exit
ELSE
 father(i) = (i + k - 2) div k
 IF priority(father(i)) < priority(i)  THEN
  BEGIN
    exchange the father(i) and A(i)
    siftup(father(i))
  END
END SIFTUP .
```

DELETION: Since the highest priority is at the root, remove it and put the last item in the root position. Move the terminate symbol zero to predecessor of it, and decrease the number of items in the k-ary tree by 1, and then call procedure SIFTDOWN to satisfy the k-ary property. Figure 9 illustrates the deletion process from figure 7.

```
PROCEDURE SIFTDOWN(m,z)
/* start with root m, and last item z. */
IF m is a leaf node THEN  exit
ELSE
  find the largest son of m
  IF priority(largest son) > priority(m)  THEN
   BEGIN
     exchange them
     siftdown(the largest son position of m,z)
   END
END SIFTDOWN .
```

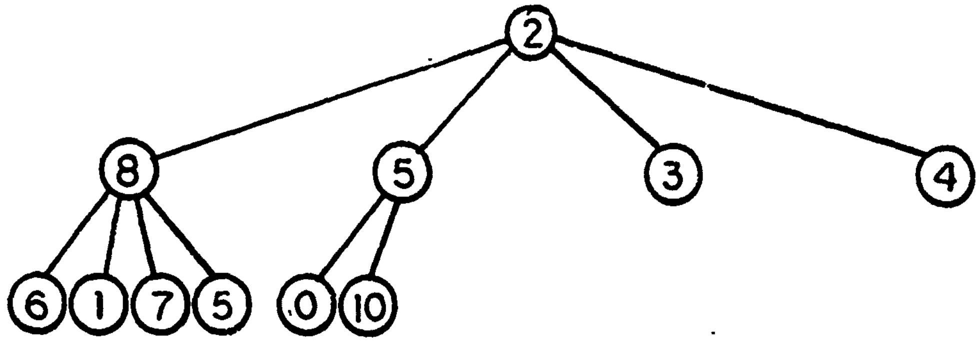


After insertion of priority 9 into fig.7 and before siftup.

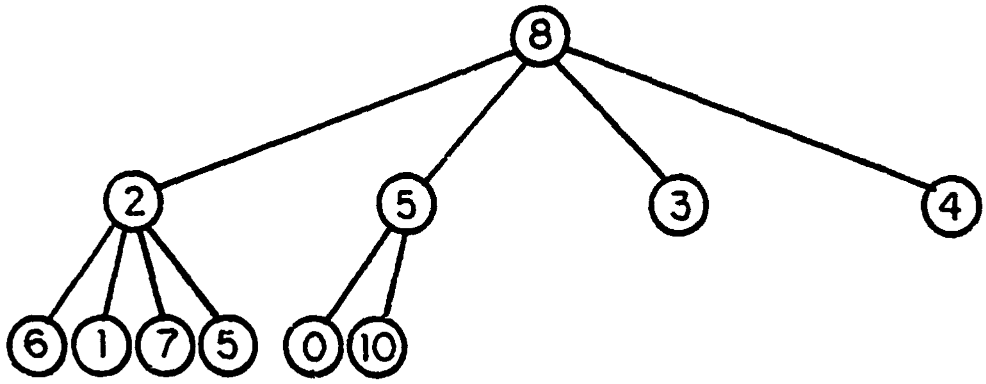


After siftup.

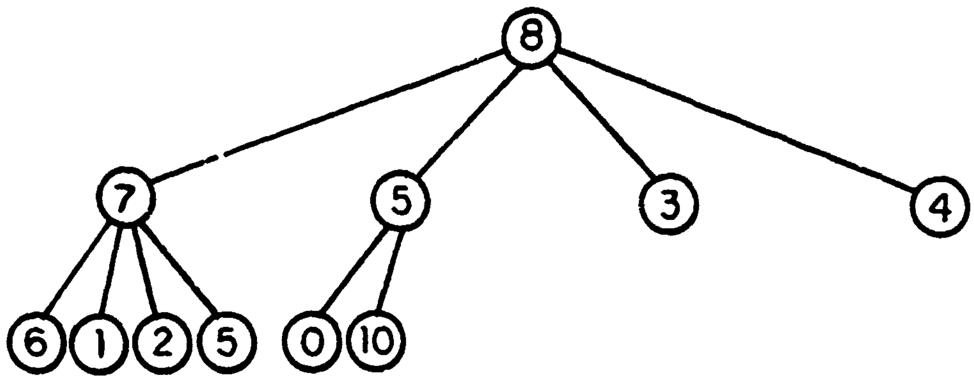
Figure 8



After deletion of highest item from fig.7 and before siftdown.



After one siftdown operation.



After second siftdown operation.

Figure 9.

LEMMA: The height of a complete k-ary tree with N nodes is equal to $\lfloor \log_k N \rfloor$.

PROOF: Since, the number of nodes on the successive levels of a complete k-ary tree follows a geometric progressions 1, k, k^2, $k^3 \ldots k^h$, the total number of nodes, N, in a complete k-ary tree of height h is bounded by,

$$\sum_{i=0}^{k-1} 3^i + 1 \quad <= N <= \quad \sum_{i=0}^{k} 3^i .$$

$$\frac{k^h - 1}{2} + 1 <= N <= \frac{k^{h+1} - 1}{2} \quad ==> \quad \frac{k^h - 1 + 2}{2} <= N <= \frac{k^{h+1} - 1}{2}$$

from this,

$$\log_k\left(\frac{k^h + 1}{2}\right) <= \log_k N <= \log_k\left(\frac{k^{h+1} - 1}{2}\right)$$

$$\log_k(k^h + 1) - \log_k 2 <= \log_k N <= \log_k(k^{h+1} - 1) - \log_k 2$$

for the left hand side of equation

$$h - \log_k 2 \quad < \quad \log_k(k^h + 1) - \log_k 2 \quad <= \quad \log_k N$$

since $h < \log_k(k^h + 1)$, also for $k >= 2$ $\log_k 2 <= 1$.

So, $h - 1 <= h - \log_k 2$

thus summing up $h - 1 < \log_k N$.

For the right hand side of equation

$$(h+1) - \log_k 2 \quad > \quad \log_k(k^{h+1} - 1) - \log_k 2 \quad >= \quad \log_k N$$

since $h+1 > \log_k(k^{h+1} - 1)$, also for $k >= 2$ $\log_k 2 <= 1$.

So, $h+1-1 > h+1-\log_k 2$

thus summing up $h + 1 > \log_k N$.

Since $h-1 < \log_k N < h+1$ clearly $h = \lfloor \log_k N \rfloor$.

K-ARY TREE INSERTION WORST CASE ANALYSIS: This analysis is very similar to the insertion worst case analysis of the heap. The only difference is the depth of the tree which would be equal to d= $\lfloor \log_k N \rfloor$ where k is the degree of the tree and N is the number of items in the tree after insertion. Bases of logarithms can be changed to base 2 by using formula;

$$\log_2 N = \frac{\log_k N}{\log_k 2}$$

K-ARY TREE DELETION WORST CASE ANALYSIS: A k-ary tree deletion algorithm does d+1 exchanges of keys (as in the heap) with depth d= $\lfloor \log_k N \rfloor$, after the deletion.

The number of key comparisons would be equal to ((k-1)+1)d=k*d because, k-1 comparisons are made between k sons in order to find the bigger son and one comparison is made between the bigger son and its father: a total of k comparisons are done at each run of the procedure siftdown and this procedure call itself recursivly at most d times. So, deletion take O(log N) time.

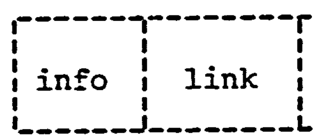
STORAGE REQUIREMENT FOR A K-ARY TREE: This method also uses an array to hold a priority of an items and storage requirement would be the same as in a heap. I.e.. N units for N nodes, and N*I units for information.

C. SINGLE LINKED-LIST

DEFINITION: A single linked list is a finite sequence of nodes such that each has a pointer field contains a pointer to the next node[13].

A pointer to the list, i.e., to the first node, is called FRONT, and a pointer to the last node, is called BACK. The priorities are arranged in decreasing order from FRONT to BACK.

IMPLEMENTATION: Each node in the linked list is of the form;



where the link field contains the pointer to the next node, and info field contains the priority of the item. Initially, the empty linked list contains only the empty pointers FRONT and BACK. Figure 10 illustrates the single linked list with 4 items in it.

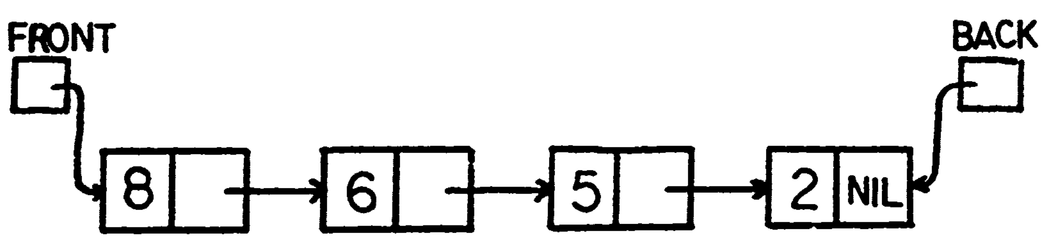


Figure 10

INSERTION: In order to add a new node X to the singly linked list, straight insertion method is used. The straight insertion involves two basic operations:

i) scanning an ordered linked list, starting at FRONT, to find the largest priority less than a priority of new item, and

ii) inserting a new item into a specified part of the ordered linked list.

As a result of insertion, the highest item will be linked to the FRONT. Figure 11 illustrates the insertion operation.

```
PROCEDURE INSERT(X)
  create a new node and initialize
  IF priority(FRONT) < priority(X) THEN
     link(X)=FRONT
     FRONT=X
  ELSE
     find the node with the largest priority less than
     the priority(X).
     IF it is reached to the BACK  THEN
        link(BACK)=X
        BACK=X
     ELSE
        let W be the pointer to the predecessor of that item
        link(X)=link(W)
        link(W)=X
END INSERT.
```

DELETION: Since the highest priority in the linked list will be pointed by FRONT, save it in somewhere, and link the FRONT to the successor of the highest item. Figure 12 illustrates the deletion operation.

```
PROCEDURE DELETE
 IF there is only one item in the queue  THEN
    FRONT=nil
    BACK=nil
 ELSE
    FRONT=link(FRONT)
END DELETE.
```

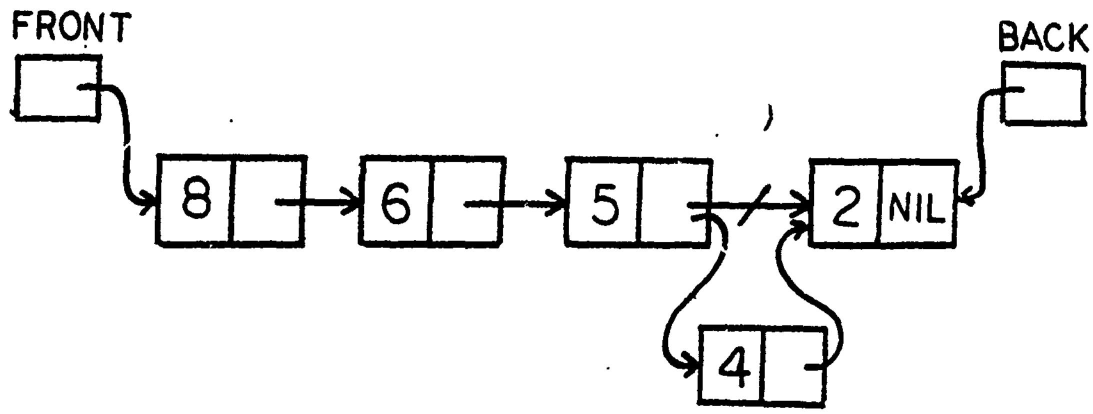


Figure 11. After insertion of 4 into fig.10.

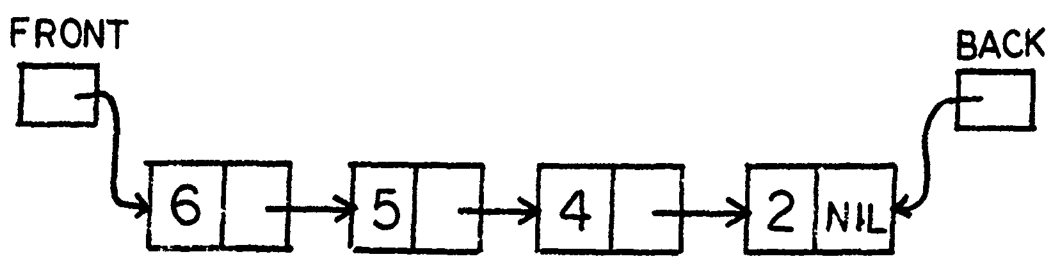


Figure 12. After deletion from fig.11.

LINKED LIST INSERTION WORST CASE ANALYSIS: The worst case occurs if the new item is smaller than the smallest key in the queue. In this case, N-1 comparisons of keys are needed in order to add the new item to single linked list, if there are N items after insertion. There would not be any exchanges of keys. So, insertion takes O(N) time.

LINKED LIST DELETION WORST CASE ANALYSIS: Deletion from single linked list takes constant time; since a pointer FRONT points to the highest key in the queue, FRONT will be linked to the successor of the highest item after deletion. There would not be any comparisons and exchanges of keys.

STORAGE REQUIREMENT FOR A SINGLE LINKED-LIST: Each node in this method contains one pointer field and one priority field. In addition to the nodes in the queue, there are two pointers which point the front and the back of the linked-list. If there are N items in the queue, required storage would be N priority fields, N+2 pointer fields, and N*I units space for the information where, I is the size of information at each node.

D. LEFTIST TREE

DEFINITION: A leftist tree is a linked binary tree with some special properties which mentioned below. It was discovered in 1971 by Clark A.Crane[2]. A leftist tree has the following advantages over a heap.

1)Insertion and deletion take O(log N) steps, and insertion and deletion take constant time in the case that insertion obey a stack discipline.

2)The records never move, only the pointers change.

3)It is possible to merge two disjoint priority queues, having a total of N elements, into a single priority queue, in only O(log N) steps. That is why leftist trees are suitable for applications where fast merging is required[4,8].

IMPLEMENTATION: The data structure record is used to implement nodes in the leftist tree. Each node is in the form;

llink	info	rlink
left	DIST KEY	right

Where the llink and rlink fields contain pointers left and right to the nodes corresponding to the left and right descendants of the node. The pointer fields are set to nil if the corresponding descendants are empty. The DIST field

is always set equal to the length of the shortest path from that node to a leaf, and the KEY field contains the priority of the item. The KEY and DIST fields in the leftist tree satisfy the following properties;

1) KEY(P) >= KEY(left(P))

   KEY(P) >= KEY(right(P))

2) DIST(P) = 1+MIN(DIST(left(P)), DIST(right(P))

3) DIST(left(P)) >= DIST(right(P))

Relation 1 is analogous to the heap condition (2) stated in definition of heap. Relation 3 implies that a shortest path to a leaf may always be obtained by moving to the right [8].

The leaf nodes in the leftist tree do not hold any information, and contain empty left and right pointers, and zero DIST and KEY fields. Figure 13 illustrates a leftist tree with 8 items in it, where the first number at each node is a KEY and second number is a DIST.

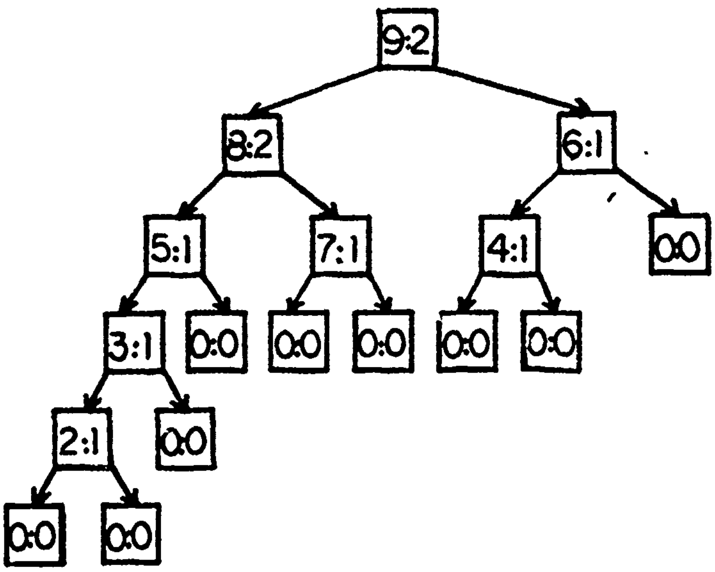


Figure 13

INSERTION: Search for the new node, starts at root P. If the KEY(P) is greater than the KEY of new item, travel is made thru subtree whose height is smaller than the other, otherwise KEY exchanges are made and then travel is made in the same fashion. The algorithm for insertion is given below. Figure 14 illustrates the insertion process.

```
PROCEDURE INSERT(PRTY,P,H)
/* Insert the new item with priority PRTY into leftist tree
with root P. H is a boolean variable and is set TRUE if P is
a leaf node. */
      IF P is a leaf node  THEN
          create a new node and set DIST field equal to 1
          set H = true
      ELSE
         IF priority(P) >= PRTY   THEN
            IF DIST(right) >= DIST(left) THEN
               INSERT(PRTY,left son of P,H)
               H=false
            ELSE
               INSERT(PRTY, right son of P,H)
               update DIST field
         ELSE
            exchange PRTY and  priority(P)
            INSERT(PRTY, P,H)
END INSERT.
```

DELETION: Since the highest priority item is always at the root, it is only necessary to remove the root and merge its two subtrees. First the bigger son of the root is made root, and then procedure MERGE is called. In merging process travel is always made thru the left branch. If it is necessary, the priority exchanges are made during travel. Figure 15 illustrates the deletion process.

```
PROCEDURE DELETE(P)
/* Remove the root P and call procedure MERGE to merge two
subtrees, pointed to respectively by R and Q. */
      IF key(R) > key(Q) THEN
          make R root
          MERGE(Q, left son of R)
      ELSE
          make Q root
          MERGE(R, left son of Q)
END DELETE.
PROCEDURE MERGE(P1, P2)
/*Merge two disjoint trees, pointed to by P1 and P2 */
      IF P2 shows leaf node   THEN   P2=P1
      ELSE
         IF key(P1) > key(P2) THEN
              exchange P1 and P2
              MERGE(P1,left son of P2)
         ELSE
              MERGE(P1,left son of P2)
END MERGE.
```

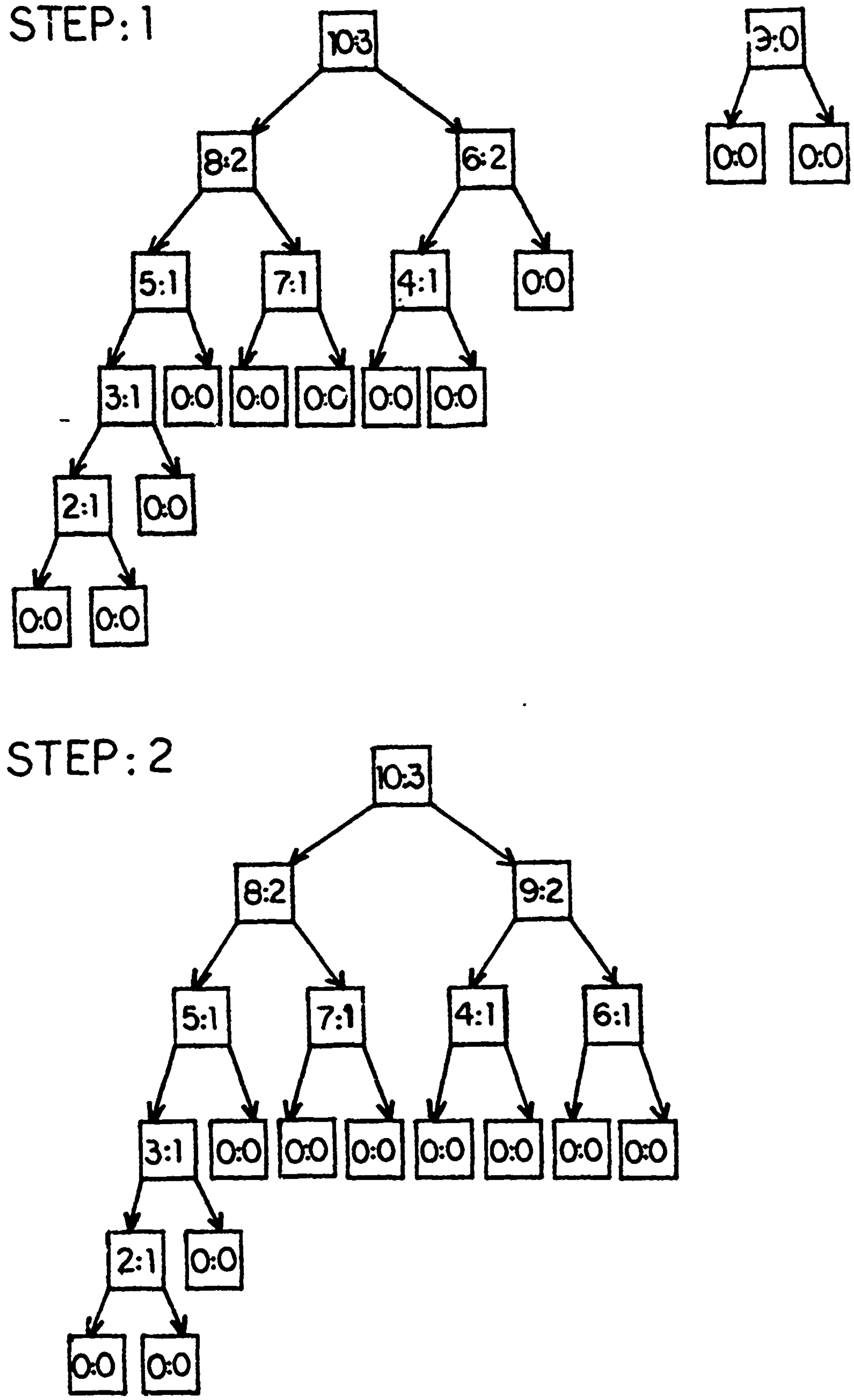


Figure 14. Insertion process of 10 into figure 13.

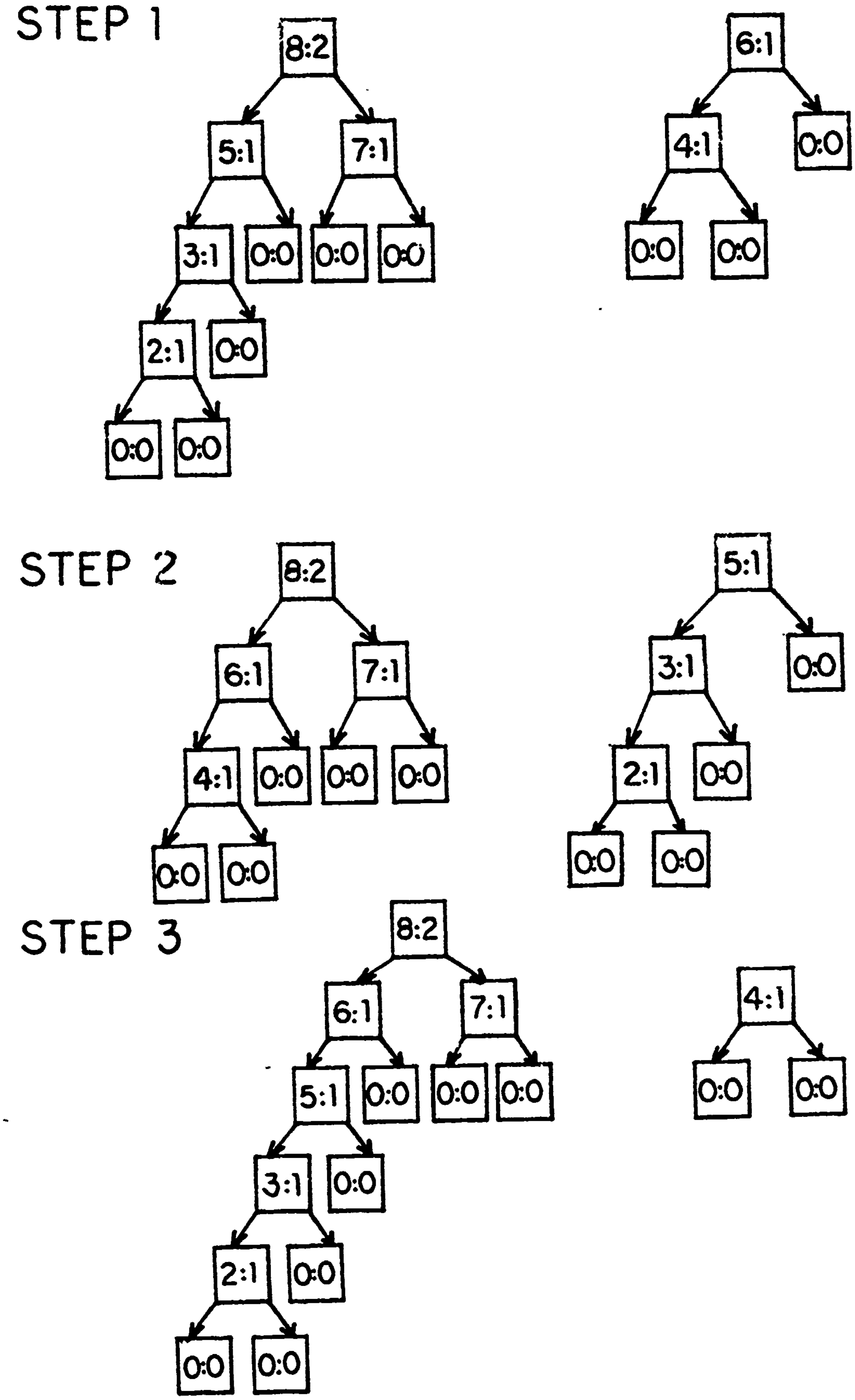
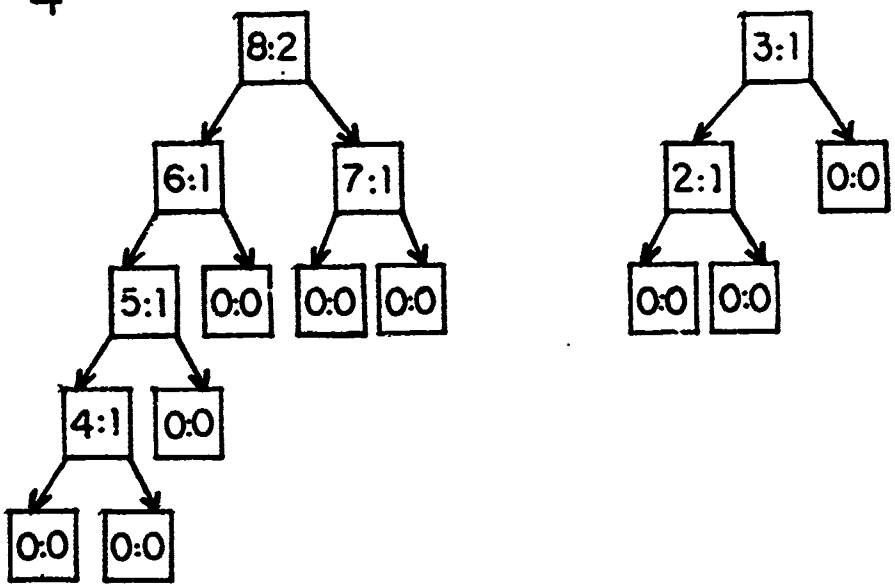


Figure 15. Deletion process from fig. 13.

STEP 4



STEP 5

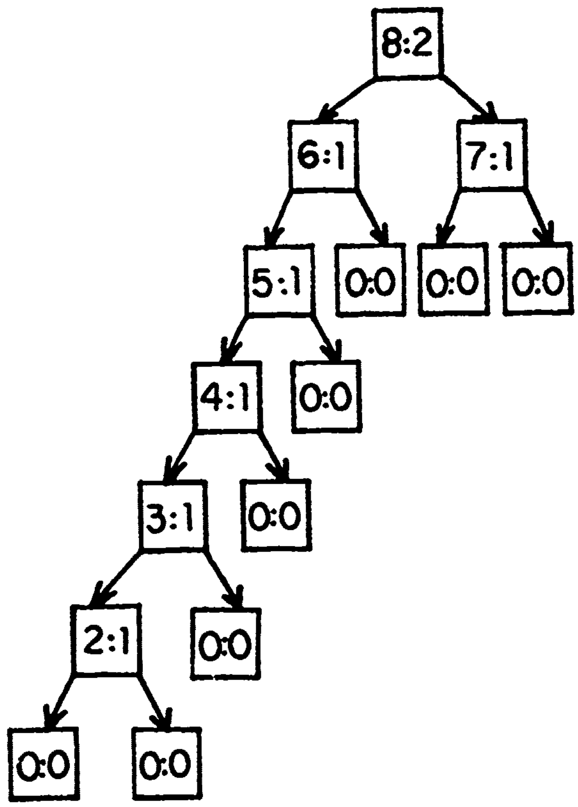


Figure 15. Deletion process from fig.13(continued).

LEFTIST TREE INSERTION WORST CASE ANALYSIS: Insertion is based on the DIST field which shows the shortest path from that node to a leaf. The search path is always choosen thru the smaller DIST field, it means thru the shortest path to a leaf.

A worst case occurs if the priority of the new item is bigger then the priority of the root and the shortest path is equal to the longest path, in which case a leftist tree would be completely balanced and the height of a tree would be $\lfloor \log_2 N \rfloor$ if there are N items in it after insertion.

There would be one key comparison at each level including level Ø and one DIST field comparisons of left and right sons; total $\lfloor \log_2 N \rfloor + \lfloor \log_2 N \rfloor = 2\lfloor \log_2 N \rfloor$. The number of key exchanges would be as many as the number of key comparisons which is equal to $\lfloor \log_2 N \rfloor$.

LEFTIST TREE DELETION WORST CASE ANALYSIS: Let's call the left son and right son of root L and R. The worst case occurs if the right subtree has only one element, whose key is the smallest in the tree and the right pointer of the nodes in the left subtree, points to the empty node. This case illustrated in figure 16. After the deletion of the highest key at the root, L would be the new root and the key of R has to be compare with all keys of the left subtree of L. The number of comparisons would be equal to N-1 if there are N items in the tree after deletion.

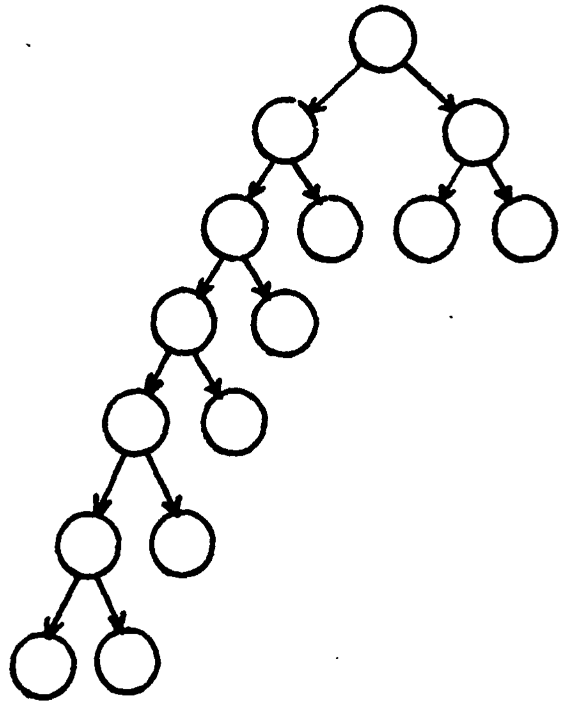


Figure 16

STORAGE REQUIREMENT FOR A LEFTIST TREE: In this implementation all information about the items is held by the internal nodes and the external nodes contain no information. If there are N items in the queue there would be N+1 empty external nodes; total 2N+1 nodes. Each node contains two pointer fields and two integer fields. The storage requirement for this method would be 4N+2 integer fields, 4N+2 pointer fields, and N*I units space for information where, I is the size of information at each node.

E. LINKED TREE

Another implementation of a priority queues can be done by using linked binary trees. Figure 18 illustrates the linked binary tree representation of the binary tree shown in figure 17.

DEFINITION: A linked tree is a binary tree with some special properties.

1) The key at each node is greater than or equal to the key at each of its son (if it has any).

2) Each node contains the number of descendants of itself. This value is set to zero if the node does not have any descendants.

3) Insertion of the new item into tree without any deletion provides completely balanced binary tree. The deletion of two or more items might destroy balance of the tree, but subsequent insertions will force the tree to be balanced.

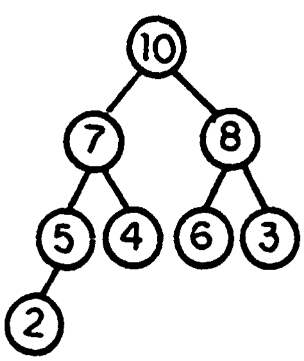


Figure 17

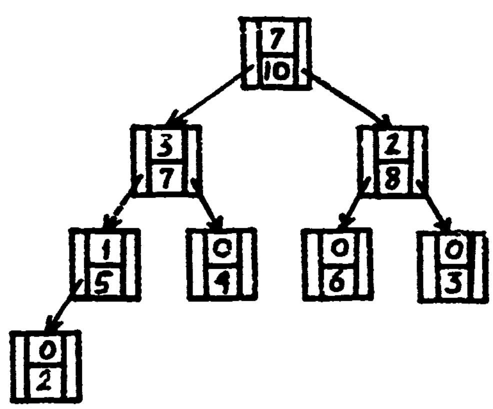


Figure 18

IMPLEMENTATION: The data structure RECORD is used to implement nodes. Each node is of the form:

```
  llink   info   rlink
+-------+-------+-------+
|       |       |       |
|       | DESC  |       |
| left  +-------+ right |
|       |       |       |
|       |  KEY  |       |
+-------+-------+-------+
```

where the llink and rlink fields contain pointers left and right to the nodes corresponding to the left and right descendants of node. The pointer fields are set to nil if the corresponding descendants are empty. The KEY field contains priority of the item, and the DESC field contains the number of descendants in the tree. Since the leaf nodes don't have any descendants, DESC field of the leaves are always set equal to zero.

INSERTION: Before calling procedure Insert, a new node is created and initialized in the main program. Insertion is mainly based on the DESC fields of the nodes. Searching started from the root and traveled thru the smaller DESC field. If the DESC fields are equal of the left and right sons, travel is made thru left branch. This method will first fill left sons and then right sons from left to right at particular level. If the key of the new item is greater than any key of the node on the travel path, only the key exchanges are made, and travel continues until it has reached the leaf nodes. Figure 19 is the illustration of insertion process.

```
PROCEDURE INSERT(W,PRTY)
/* insert the new key PRTY, into linked tree with root W */
 IF key(W) >= PRTY  THEN
  BEGIN
   IF left(W) = nil  THEN
      link new item to it
   ELSE IF right(W) = nil  THEN
      link new item to it
   ELSE IF DESC(left(W)) > DESC(right(W))  THEN
      BEGIN
       W:=right(W)
       DESC(W):=DESC(W) + 1
       INSERT(W,PRTY)
       END
    ELSE
      BEGIN
       W:=left(W)
       DESC(W):=DESC(W) + 1
       INSERT(W,PRTY)
      END
  END
 ELSE
  BEGIN
   exchange key(W) and PRTY
   INSERT(W,PRTY)
  END
END INSERT.
```

DELETION: Since the highest key at the root, only remove key field of the the root and move the bigger son's key to the root, and travel thru the moved son. This process continues until it has reached the leaf nodes. There is not any rebalancing process. The deletion process is illustrated in figure 20.

```
PROCEDURE DELETE(X)
/* delete the key(root), without deleting root, and siftup
   the bigger son's key and travel thru the moved son. */
IF key(left(X)) > key(right(X))  THEN
   BEGIN
    move key(left(X)) to key(X)
    decrease the DESC field of left(X) by 1
    IF it has reached  the leaf node  THEN  exit
    ELSE delete(left(X))
   END
ELSE
   BEGIN
    move key(right(X)) to  key(X)
    decrease the DESC field of right(X) by 1
    IF it has reached  the leaf node  THEN  exit
    ELSE delete(right(X))
   END
END DELETE.
```

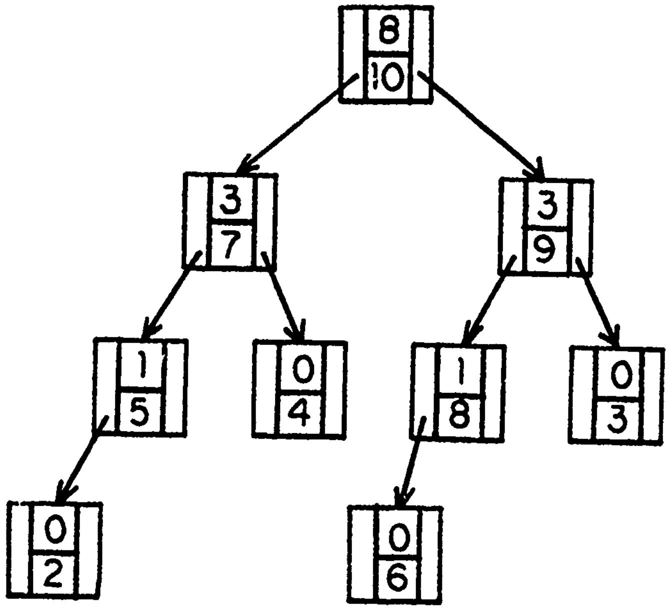


Figure 19. After insertion of 9 into fig.18.

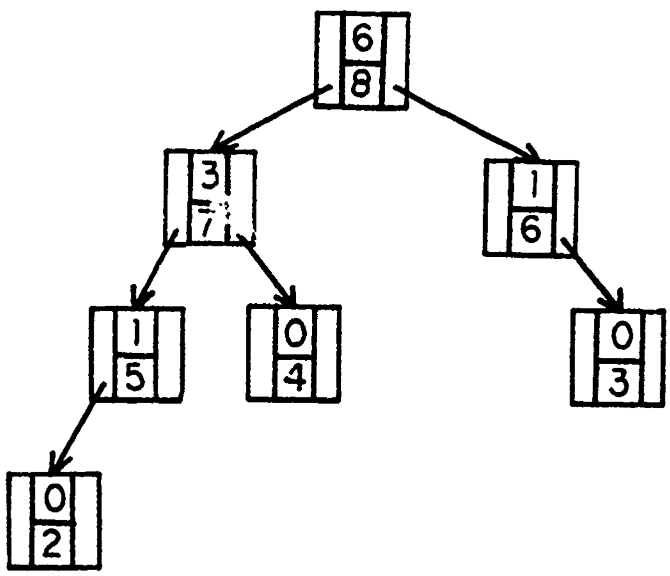


Figure 20. After deletion from fig.18.

LINKED TREE INSERTION WORST CASE ANALYSIS: In this method, insertion based on the DESC field of the nodes and travel is always made thru the smaller DESC field, it means always the shortest path (from root to the leaf) is traversed during an insertion process. The worst case occurs if the priority of the new item is bigger than the priority of the root and the shortest path is equal to the longest path, in which case a linked tree would be completely balanced, and the height of a tree would be $\lfloor \log_2 N \rfloor$ if there are N items in it after insertion.

There would be one key comparisons at each level, including level 0 and one DESC's field comparisons of left and right sons: total $\lfloor \log_2 N \rfloor + \lfloor \log_2 N \rfloor - 1 = 2\lfloor \log_2 N \rfloor - 1$. The number of key exchanges would be as many as the number of key comparisons which is equal to $\lfloor \log_2 N \rfloor$.

LINKED TREE DELETION WORST CASE ANALYSIS: Since there is not any rebalancing process in the linked tree, deletion from the tree could unbalance it.

The worst case occurs if there are two items at each level in the tree (except level zero), and small priority at each level does not have any descendants. This worst case situation is illustrated in figure 21 and the bigger item at each level has filled with cross sign. In the worst case the height of the tree would be N/2 if there are N items in the queue before deletion. Since there is one key comparison and one key exchange at each level, total N/2 key comparisons

and key exchanges are needed after the deletion of priority at the root.

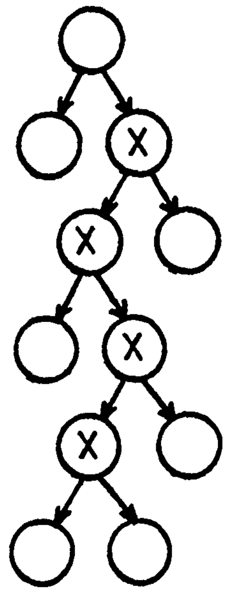


Figure 21

STORAGE REQUIREMENT FOR A LINKED TREE: Each node contains two pointer fields and two integer fields in this implementation. If there are N items in the queue, required storage would be 2N pointer fields, 2N integer fields, and N*I units space for the information where I is the size of information at each node.

F. AVL TREE

Adelson-Velskii and Landis in 1962 introduced a binary tree structure that is balanced with respect to the heights of subtrees[9]. The height of a tree is defined to be its maximum level, the length of the longest path from the root to an external node.

DEFINITION: A binary tree is called balanced if the height of the left subtree of every node never differs by more than +1 or -1 from the height of its right subtree[9]. As a characteristic of AVL tree, the priority of the left son is smaller and priority of the right son is bigger than its parent's priority. Trees satisfying above definition are often called AVL-TREES after their inventors.

As a result of the balanced nature of this type of tree, dynamic retrievals can be performed in O(log N) time if the tree has N nodes in it. A new node can be entered or the node with highest priority can be deleted from such a tree in time O(log N). The resulting tree remains height balanced. Figure 1 shows an AVL-TREE with N=7, and node structure.

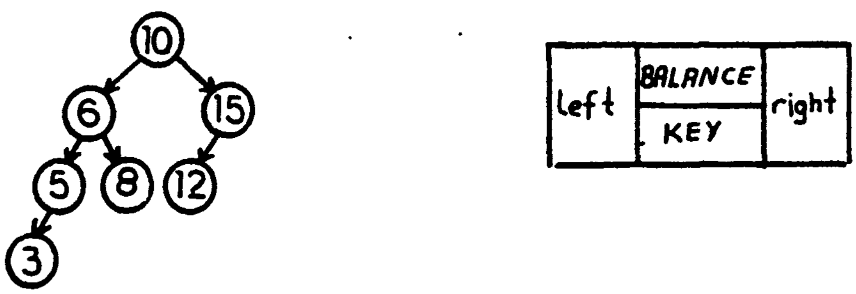


Figure 22

IMPLEMENTATION: Each node in the tree contains a KEY field containing the priority, a LEFT and RIGHT pointers which point to the corresponding left and right descendants of a node. The BALANCE field which may have either -1, 0, or +1 to indicate the differences of the height of the left and right subtrees. If L and R indicate the left and right subtrees of the node P respectively, then the BALANCE factor at P has the following meaning:

BALANCE(P) = -1 : height(R) = height(L) - 1

BALANCE(P) = 0 : height(R) = height(L)

BALANCE(P) = +1 : height(R) = height(L) + 1

INSERTION: In order to insert a node with priority X into an AVL tree, the proper place has to be found by making search. Search starts from root, P by comparing KEY(P) with X. If the new item is less than the KEY(P), travel is made thru the left branch, otherwise thru the right branch until it is reached to the leaf node. An insertion of a new item to an AVL-TREE, given a root P with the left and right subtrees L and R, causes tree to have different cases. If the new node is inserted in the left subtree and caused its height to increase by 1 ;

1)height(L) = height(R):L and R become of unequal height, but the balance criterion is not violated.

2)height(L) < height(R):L and R obtain equal height.

3)height(L) > height(R):The balance criterion is violated and tree must be rebalanced [5].

The rebalancing is carried out using essentially four

different kinds of rotation LL, RR, LR, and RL. If rebalancing is necessary after the insertion, only one of these rotations will be sufficient to rebalance the tree. These rotations are characterized by the nearest ancestor, A, of the inserted node X, whose balance factor was already +1 or -1. The following characterization of rotation types is obtained.

LL: The new node X is inserted in the left subtree of the left subtree of A, where A is the nearest ancestor of the node X, whose balance factor was already -1. The algorithm and example is shown below.

```
LL ROTATION ;
/* Balance(P) = -1, and balance(left(P)) = -1, where P
   is the pointer to A */
   p1:=left(P)
   left(P):=right(P1)
   right(P1):=P
   balance(P):=0  and  P:=P1
END LL ROTATION.
```

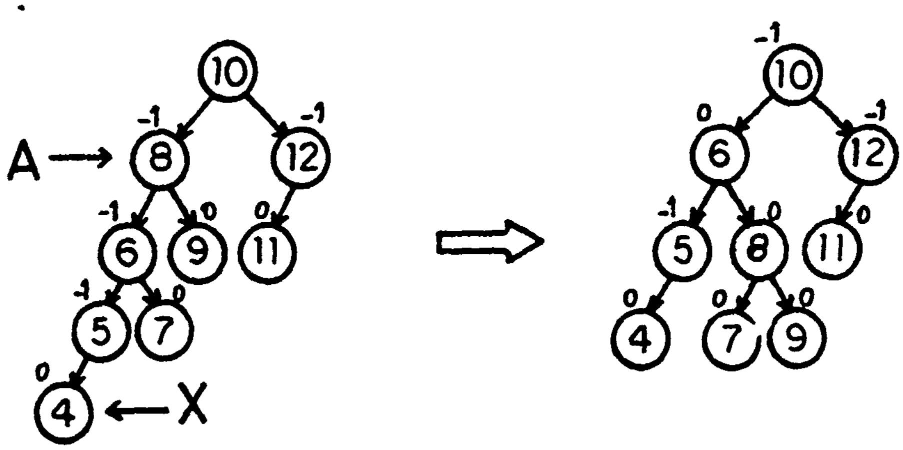


Figure 23 LL Rotation on P.

LR: X is inserted in the right subtree of the left subtree of A, where A is the nearest ancestor of the node X, whose balance factor was already -1. The algorithm and example is shown below.

```
LR ROTATION ;
/* balance(P) = -1 and balance(left(P)) <> -1, where P
  is the pointer to A. */
  P1:=left(P)
  P2:=right(P1)
  right(P1):=left(P2)
  left(P2):=P1
  left(P):=right(P2)
  right(P2):=P
  readjust balance(P) and balance(P1)
  P:=P2
END LR ROTATION.
```

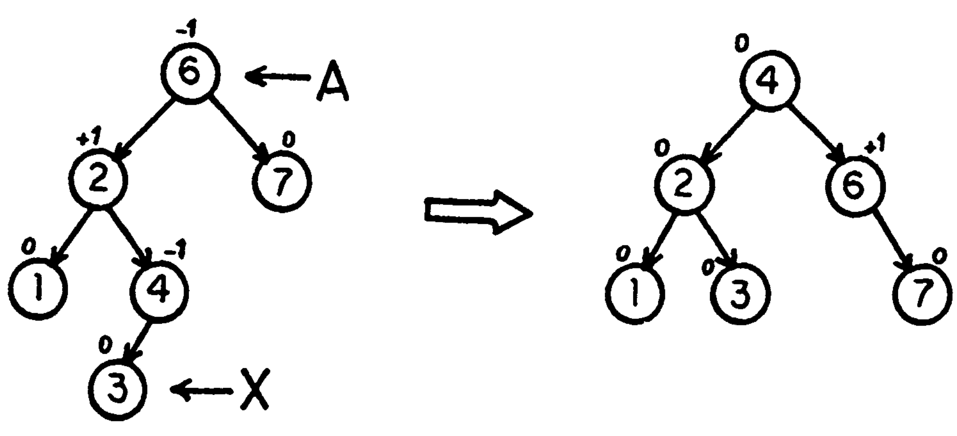


Figure 24 LR Rotation on P.

RR: X is inserted in the right subtree of the right subtree of A, where A is the nearest ancestor of the node X, whose balance factor was already +1. The algorithm and example is shown below .

```
RR ROTATION ;
/* balance(P) = +1 and balance(right(P)) = +1, where P
   is the pointer to A */
  P1:=right(P)
  right(P):=left(P1)
  left(P1):=P
  balance(P):=0
  P:=P1
END RR ROTATION.
```

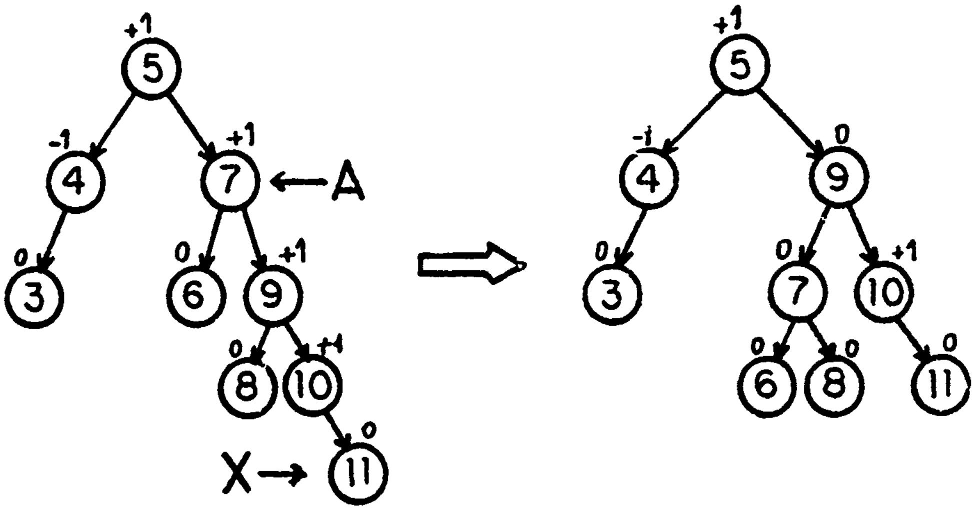


Figure 25 RR Rotation on P.

RL: X is inserted in the left subtree of the right subtree of A, where A is the nearest ancestor of the node X, whose balance factor was already -1. The algorithm for RL rotation and example is shown below.

```
RL ROTATION ;
/* balance(P) = +1  and  balance(right(P)) <> +1, where  P
   is the pointer to  A. */
  P1:=right(P)
  P2:=left(P1)
  left(P1):=right(P2)
  right(P2):=P1
  right(P):=left(P2)
  left(P2):=P
  readjust balance(P) and balance(P1)
  P:=P2
END RL ROTATION.
```

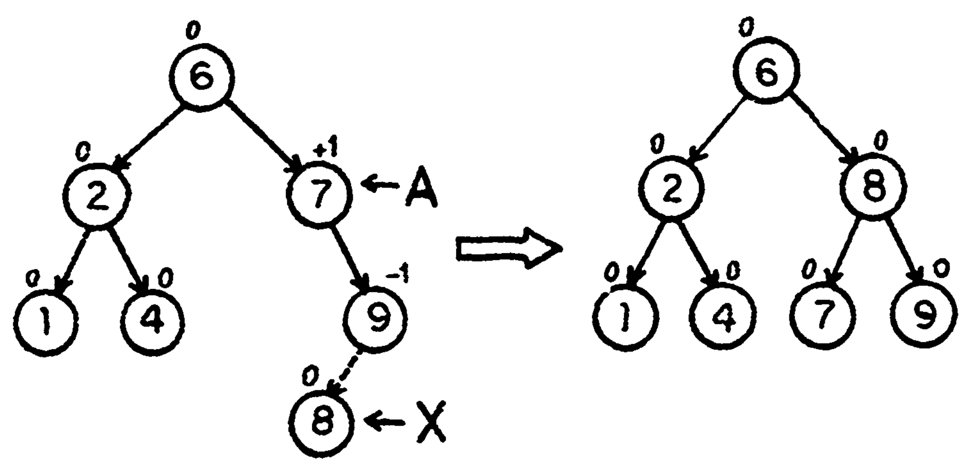
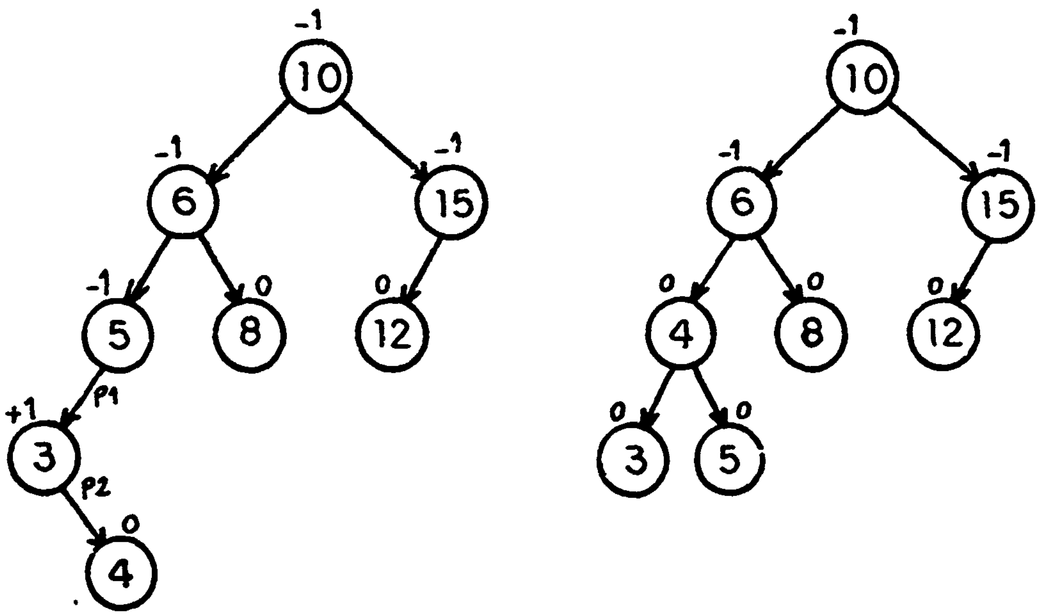


Figure 26 RL Rotation on P.

```
PROCEDURE INSERT(X,P,H)
/* The new item X is inserted into the AVL-TREE with root P.
H is true iff the subtree height has increased. */
    IF P is leaf node  THEN
       create new node and initialize.
       set H true.
    ELSE
       IF  X < key(P)  THEN
          INSERT(X,left son of P,H)
          IF  H=true  THEN
              CASE balance(P)  OF
         0:balance(P)=-1
         1:balance(P)=0 and  H=false
        -1:IF balance(left(P)) = -1  THEN
               do LL rotation on P
               balance(P)=0
            ELSE
               do LR rotation on P
               update balance(P) and balance(left(P))
          balance(P)=0 and  H=false
          ELSE  do nothing. /* H=false */
       ELSE
          INSERT(X,right son of P,H)
          IF H=true  THEN
             CASE balance(P) OF
         0:balance(P)=1
        -1:balance(P)=0 and  H=false
```

```
1: IF  balance(right(P)) = 1  THEN
           do  RR rotation on P
           balance(P)=0
       ELSE
           do  RL rotation on P
           update balance(P) and balance(right(P)).
   balance(P)=0 and  H=false
   ELSE  do nothing /* H=false */.
END INSERT.
```



After insertion the priority 4 into figure 22 and before rebalancing

After rebalancing

Figure 27

DELETION: A deletion from an AVL-TREE could unbalance it. The rebalancing operation remains essentially the same as for insertion. Since the highest priority is always at the rightmost node position, only LL and LR rotations are needed to rebalance the tree. A boolean variable parameter H has the meaning "the height of the subtree has been reduced." Rebalancing has to be considered only if H is true.

```
PROCEDURE DELETE(P,H)
/* Start with root P and travel thru the right son until the
rightmost node is found. Remove it and rebalance the tree by
traveling back thru the root. */
      find the rightmost node and delete it
      travel back thru the root , and
      call BALANCE(P,H) if necessary.
END DELETE.



PROCEDURE BALANCE(P,H)
/* This routine is called if H=true; the right branch has
become less high. */
      CASE balance(P)  OF
   1: balance(P)=0
   0: balance(P)=-1  H=false
  -1: IF balance(left(P))  <=  0  THEN
          do  LL rotation
```

```
        update balance(P) and balance(left(P)).
    ELSE
        do  LR rotation
        update balance(P) and balance(left(P)).
END BALANCE.
```

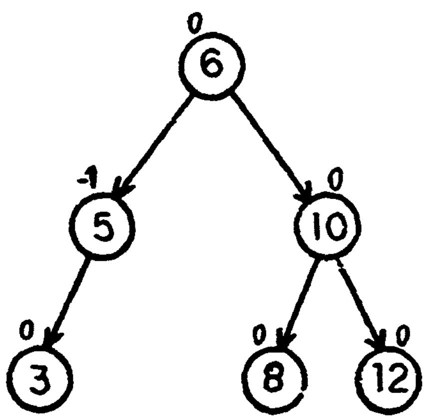


Figure 28 After deletion of highest item and LL rotation from figure 22.

AVL-TREE INSERTION WORST CASE ANALYSIS: In order to find the maximum height of an AVL tree with N nodes, consider a fixed height(h) and try to construct the AVL tree with the minumum number of nodes. The following analysis appears in reference[6].

First of all, let's take N nodes and attempt to arrange them to produce the AVL tree of greatest depth. The idea is, systematically to favor the right(left) subtree by using the least number of nodes to create the left(right) subtree of height (h-2) and the least possible number to produce a right(left) subtree of height (h-1). As a result, if we include the root, the height of the tree would be h.

Since the balance property of an AVL tree must hold for all subtrees of an AVL tree, similar conditions must hold recursively for the left and right subtrees. Figure 29 illustrates a sequence of such right-leaning AVL trees of deepest extent for N nodes.

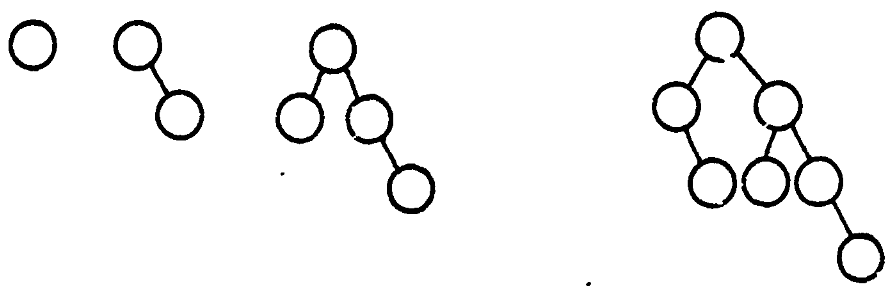


Figure 29.

The number of nodes in the left and right subtrees of the trees in figure 29 are given in table 1.

height	number of nodes in left subtree	number of nodes in right subtree	number of nodes in whole tree	fibonacci numbers
0	0	0	1	0
1	0	1	2	1
2	1	2	4	1
3	2	4	7	2
4	4	7	12	3
5	7	12	20	5

Table 1

It is easy to see that there is a recurrence relation that characterizes the numbers in each of the columns of this table, namely:

$$G_h = 1 + G_{h-1} + G_{h-2} \qquad \text{where, } G_0 = 0 \text{ and } G_1 = 1.$$

This recurrence seems to be a close relative of the recurrence relation for the Fibonacci sequence (The Fibonacci numbers are a sequence of integer defined by the recurrence relation $F_i = F_{i-1} + F_{i-2}$ for $i > 1$, with boundary conditions $F_0 = 0$ and $F_1 = 1$)

In fact, comparing the Fibonacci sequence

0,1,1,2,3,5,8,13,21......

to the numbers in the columns of Table 1 suggests that G_h is just one less than some corresponding Fibonacci number $G_h = F_{h+3} - 1$ for this reason, the trees of Fig 29 are called

Fibonacci trees. Since these Fibonacci trees have the fewest nodes among all possible AVL trees of height h, as indicated before the number of nodes in any AVL tree of height h obeys the relation $N >= G_h$; so,

$$N >= F_{h+3} - 1.$$

But the k'th Fibonacci number F_k is bounded below by a power of the inverse of the 'golden ratio' $\emptyset = (1+\sqrt{5})/2$. It is known that $F_k >= \emptyset^{k-2}$ and more precisely $F_k > \emptyset^k/\sqrt{5} - 1$

Hence, one can conclude

$$N > \emptyset^{h+2}/\sqrt{5} - 2 = N + 2 > \emptyset^{h+2}/\sqrt{5}$$

from this,

$$\log_\emptyset(N+2) > \log_\emptyset(\emptyset^{h+2}/\sqrt{5}) = \log_\emptyset \emptyset^{h+2} - \log_\emptyset\sqrt{5} = h + 2 - \log_\emptyset\sqrt{5}$$

$$\log_\emptyset(N + 2) + \log_\emptyset\sqrt{5} > h + 2$$

so, $h < \log_\emptyset(N+2) + \log_\emptyset\sqrt{5} - 2.$

now, using the fact that $\emptyset = 1.618034$ and applying a logarithm base conversion

$$\log_\emptyset x = (\log_2 X / \log_2 \emptyset),$$

one gets,

$$h < \frac{\log_2(N+2)}{\log_2 1.618034} + \frac{\log_2\sqrt{5}}{\log_2 1.618034} - 2$$

$$h < 1.4404\log_2(N+2) - 0.328$$

The worst case occurs if the key of new item is bigger than the biggest key in the AVL-tree, which in this case insertion would be done to the rightmost node and would cause to increase the height of the right subtree of every ancestor on the path by one. (if an AVL tree is

left-leaning, the worst case occurs if the smallest key is inserted) to restore the lost balance property exactly one of the four rotations will be sufficient. As a summary total $1.44\log_2(N+2)$ times key comparisons and one rotation need to be done in the worst case.

AVL-TREE DELETION WORST CASE ANALYSIS: Since the highest item is at the rightmost position, there would not be any key comparisons in order to find the highest key in the AVL-tree.

Deletion of the highest item may require a rotation at every node along the search path. Consider, for example, the left-leaning Fibonacci tree (opposite of the Fig. 29), the deletion of the rightmost node would require a rotation at every node along the search path, which would be done at most $\lfloor \log_2 N \rfloor$ times.

STORAGE REQUIREMENT FOR AN AVL-TREE: Each node contains two pointer fields and two integer fields in this implementation. If there are N items in the queue, required storage would be 2N pointer fields, 2N integer fields, and N*I units space for information where, I is the size of information at each node.

G. 2-3 TREE

Another implementation of a priority queues can be done by using a 2-3 tree's property.

DEFINITION: A two-three tree is a tree in which each vertex which is not a leaf node, has two or three sons, and every path from the root to a leaf is of the same length. The tree consisting of a single vertex is also a two-three tree. Figure 30 illustrates two different 2-3 trees structure.

At each vertex X which is not a leaf, there are two additional pieces of information, L and M. L is the largest element of the subtree whose root is the leftmost son of X. M is the largest element of the subtree whose root is the second son of X. All information about the priorities are at the leaf level and in increasing order from left to right. The values of L and M attached to the vertices enable one to start at the root and search for an element in a manner analogous to binary search [1].

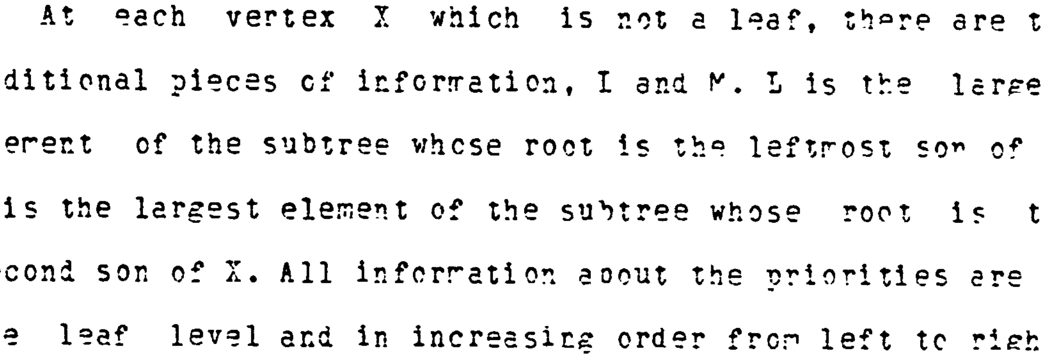


Figure 30

IMPLEMENTATION : Each node in the 2-3 tree is in the form:

```
+-------------------------------------+
|    L     |      KEY    |       M    |
+------------------------------------+
|       |        |        |         |
| left  | middle | parent |  right  |
|       |        |        |         |
+------------------------------------+
```

where left, middle, and right are the pointers to the nodes corresponding to the left, middle, and right descendants of node. The parent field is a pointer to the father of that node. Since each internal node (vertex) has 2 or 3 sons, left and right pointer fields of the internal nodes can not be empty. For the nodes which have 2 sons, the middle pointer fields will be nil. The parent field of each node, except a root can not be nil. Because only a root does not have father. The integer fields L and M contain the informations as mentioned above. The KEY field contains the information about the priority of item. The fields of L and M of the leaf nodes are always set to zero and left, middle, and right pointers are always set equal to nil. The KEY field of the internal nodes(vertices) is always set equal to zero, because the vertices are not the item, but only the node to construct the path from the root to the leaf nodes to find the expected item.

INSERTION: In order to insert the new item into 2-3 tree, the proper place has to be found by means of function SEARCH. This function starts making search with the root R, and the priority of new item PRTY. If PRTY is less than L(R) then travel is made thru the left branch, if PRTY is between L(R) and M(R), and the vertex R has 3 sons travel is made thru the middle branch, otherwise thru the right branch. This search continues until it has reached the leaf nodes. The pointer F which points to the father of these leaf nodes is returned to the calling procedure.

If that vertex F has already two sons then make the new item the appropriate son of F, and readjust the values of L and M along the path from F to the root. If F has already three sons then make the new item the appropriate son of F and call procedure ADDSON to incorporate F and its four sons into 2-3 tree. After the insertion process, the highest priority will be always at the rightmost position of the 2-3 tree. The algorithm for function SEARCH and procedure ADDSON have been given below. The insertion is illustrated in figure 31.

```
FUNCTION SEARCH(PRTY,R)
/* Where R is the root, PRTY is the key of new item */
IF any son of R is a leaf THEN return R
ELSE
 IF PRTY <= L(R) THEN search(PRTY,left(R))
 ELSE
  IF R has two sons or PRTY <= M(R) THEN
     search(PRTY,middle(R))
  ELSE   search(PRTY,right(R))
END SEARCH .


PROCEDURE ADDSON(Z)
/* This procedure takes 2-3 tree with vertex Z , which has
   four sons and converts it into 2-3 tree to satisfy the
   2-3 tree property by creating the new vertices. */
BEGIN
  create a new vertex X
  make the two rightmost sons of Z the left
   and right sons of X .
  IF Z has no father  THEN
    create a new root R
    make Z the left son and X the right son of R
  ELSE
    let F be the father of Z
    make X a son of F immediately to the right of Z
    IF F now has 4 sons THEN  addson(F)
END ADDSON .
```

DELETION: This process is the reverse of the manner by which an element is inserted. Procudure DELETE finds the rightmost item in the queue, and disconnect the pointer to that item. Let F be the father of that item. We can have three different cases in deletion process.

CASE 1 : If F is root then remove F.

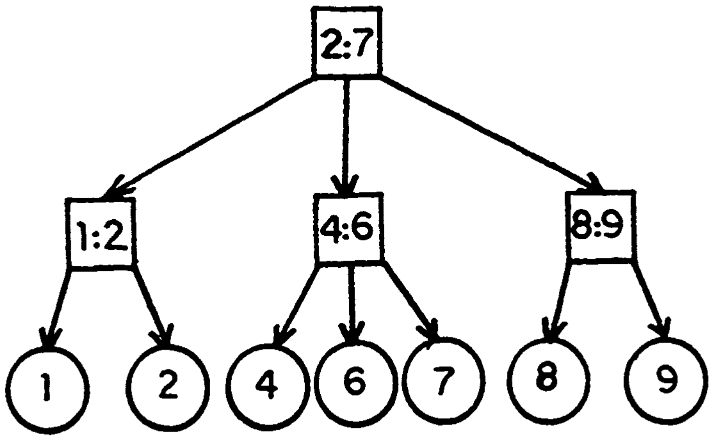
CASE 2 : If F has three sons, remove item, now F has two sons. Adjust L and M values along the path from F to root.

CASE 3 : If F has two sons , there are two possibilities. Part (b) is handled by procedure SUBSON.

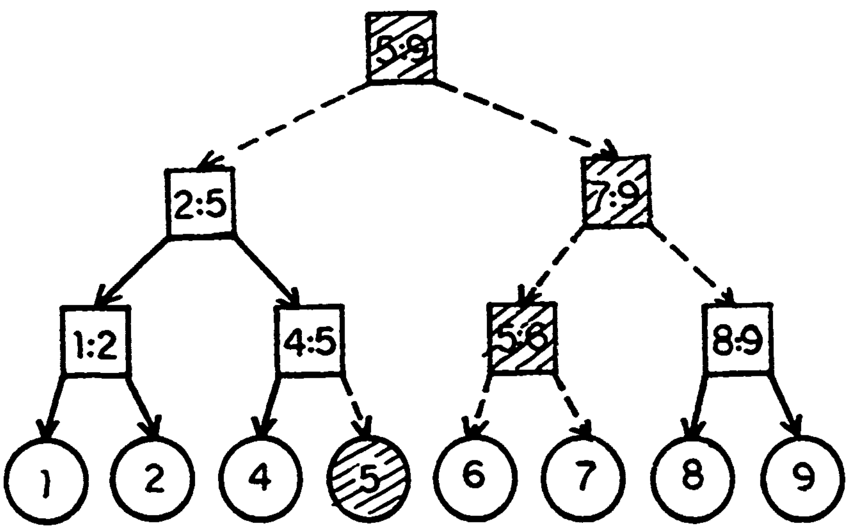
(a): If F is root, remove item and F, and leave the remaining left son as the root .

(b): F is not root; find left brother of F and call it J. If J has three sons, make the right son of J the left son of F, and adjust the L and M values of all ancestors. In this case there is not any vertex deletion. This is illustrated in figure 32(a). If J has two sons, make the left son of F the right son of J. If J is the middle son of its father, just make J right son of its father, and adjust the L and M values. This case is illustrated in figure 32(b). If J is the left son of its father, it means the father now has only a left son; find the grand-father of J and call SUBSON to incorporate J and its father into 2-3 tree[1]. This case illustrated in figure 32(c).

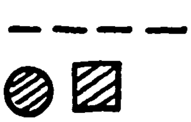
```
PROCEDURE SUBSON
/* F is the pointer to the vertex, whose right son is the
 highest item, and the middle pointer is empty.*/
IF father of F has 2 sons  THEN
   let J be left brother of F
   IF J has 3 sons  THEN
     right(F):=left(F)
     left(F):=right(J)
     right(J):=middle(J)
     adjust L and M values thru root
   ELSE
     middle(J):=right(J)
     right(J):=left(F)
     remove F and F:=parent(F)
     IF F is root THEN  root:=left(root)     ELSE subson
ELSE
   let J be left brother of F
   IF J has 2 sons  THEN
     middle(J):=right(J)
     right(J):=left(F)
     adjust L and M values thru root
   ELSE
     right(F):=left(F)
     left(F):=right(J)
     right(J):=middle(J)
     adjust L and M values thru root
END SUBSON .
```



2-3 tree before insertion.



2-3 tree after insertion 5.



new links

new created nodes

Figure 31.

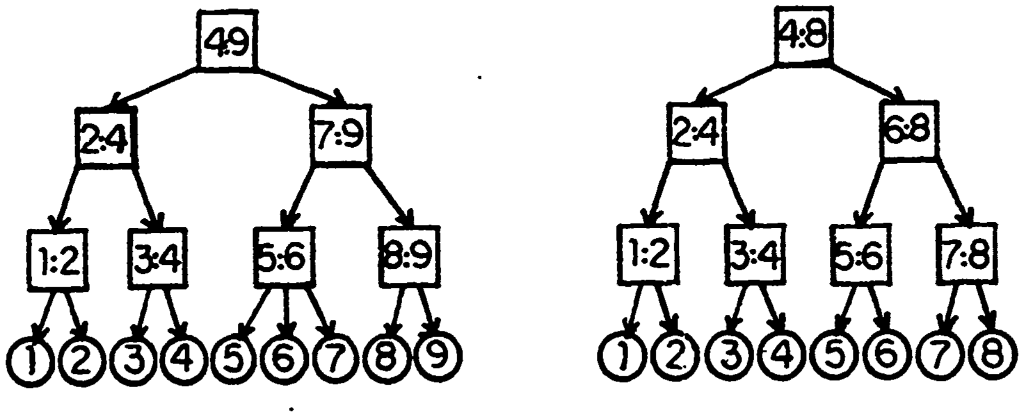


Figure 32(a). Example for case 3 part 1 of deletion.

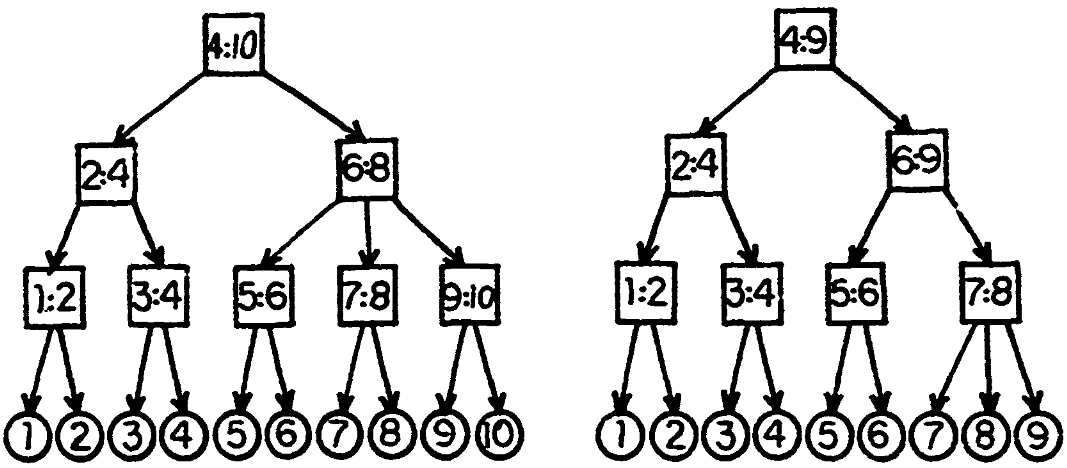


Figure 32(b). Example for case 3 part 2 of deletion.

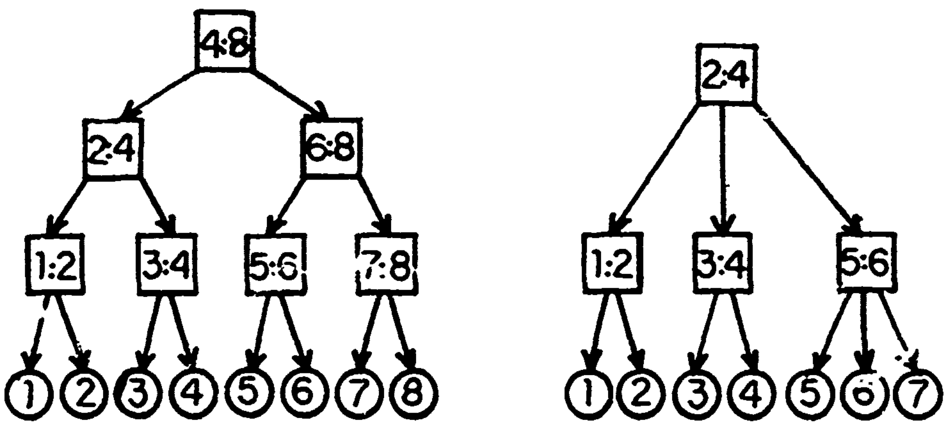


Figure 32(c). Example for case 3 part 3 of deletion.

2-3 TREE INSERTION WORST CASE ANALYSIS: It is necessary to analyze a 2-3 tree in two different ways. In this method there would not be any key exchanges, but only the update of L and M values. Two different worst cases are obtained as following:

1) The deepest 2-3 tree on N keys will be constructed by taking the minumum number of children (two) allowed for each node. So, the height of a 2-3 tree with N leaves is at most $d=\lfloor \log_2 N \rfloor$. The correct position for the newly inserted item is found by function SEARCH. In this function, the key of the new item is compared with the values of vertices. The worst case occurs if the comparisons are made with both L and M at each vertex along the path from root to the leaf. This case happens if the key of the new item is bigger than the second biggest key in the 2-3 tree. Hence the function SEARCH calls itself recursively d times. So, the total number of key comparisons would be $2\lfloor \log_2 N \rfloor$.
Since in this analysis every vertex has two sons, the newly inserted item would be the third son of the correct vertex and we would not need procedure ADDSON.

2) The worst case for the function SEARCH which mentioned in case (1), would be also same for this case. The only difference is the height of the tree, namely the total number of key comparisons would be equal to $2\lfloor \log_3 N \rfloor$. Since each vertex has 3 sons, after the insertion process, as many as d nodes have to be split as the split progresses up to the root. This is done by procedure ADDSON. There would not

be any key comparisons, but it is necessary to update L and M values along the path, from the second bottommost level to the root. In either case O(log N) is the worst case time for an insertion.

2-3 TREE DELETION WORST CASE ANALYSIS: The height of the deepest 2-3 tree would be $d=\lfloor \log_2 N \rfloor$ on N keys as mentioned earlier.

The worst case occurs if each vertex has two sons. Because, after the deletion of the highest key, the father of the highest key would have only one son left. In order to incorporate the left brother of the highest key into 2-3 tree, procedure SUPSON has to call itself d times. There would not be any key comparisons, exchanges, and update of the L and M values.

STORAGE REQUIREMENT FOR 2-3 TREE: In this method, all information about the items are held by the external nodes. Each node contains four pointer fields and three integer fields.

The maximum number of nodes would be needed if each node has two sons. In this case if there are N items in the queue, N-1 internal nodes are needed; total 2N-1 nodes. That would be 8N-4 pointer fields and 6N-3 integer fields.

The minumum number of nodes would be needed if each node has 3 sons. In this case if there are N items(external node)

in the queue, the height of the tree would be equal to $k=\lfloor \log_3 N \rfloor$. Since, the number of nodes on the successive levels of a 2-3 tree with 3 sons of each node follows a geometric progression 1, 3, 3^2, 3^3, ... 3^k, the total of nodes in the tree would be equal to,

$$\sum_{i=0}^{k} 3^i = \frac{3^{k+1} - 1}{2}$$

since, all items would be at level k, N is equal to 3^k. The number of internal nodes can be calculated using above formula;

$$\sum_{i=0}^{k-1} 3^i = \frac{3^k - 1}{2} = \frac{N - 1}{2}$$

So, the total of nodes in the tree with N external nodes would be equal to,

$$N + \frac{N - 1}{2} = \frac{2N + N - 1}{2} = \frac{3N - 1}{2}$$

that would be 6N-1 pointer fields, $\frac{9N - 1}{2}$ integer fields, and N*I units of storage where, I is the size of information at each node.

E. FIXED PRIORITY

This method of priority queue representation was discovered by Luther C.Abel [Ph.D.thesis university of Illinois 1972].

DEFINITION: In this method, all the elements of a priority queue are known to be contained in some fixed set $\{ K1, K2, \ldots KN \}$, where $K1 < K2 < K3 < \ldots\ldots < KN$.

The idea is to use the complete binary tree with N external nodes, which are implicitly associated with the keys in increasing order, from left to right[8]. Figure 33(a) shows the empty priority queue with the priority range from 1 to 7, figure 33(b) shows with 4 items in it.

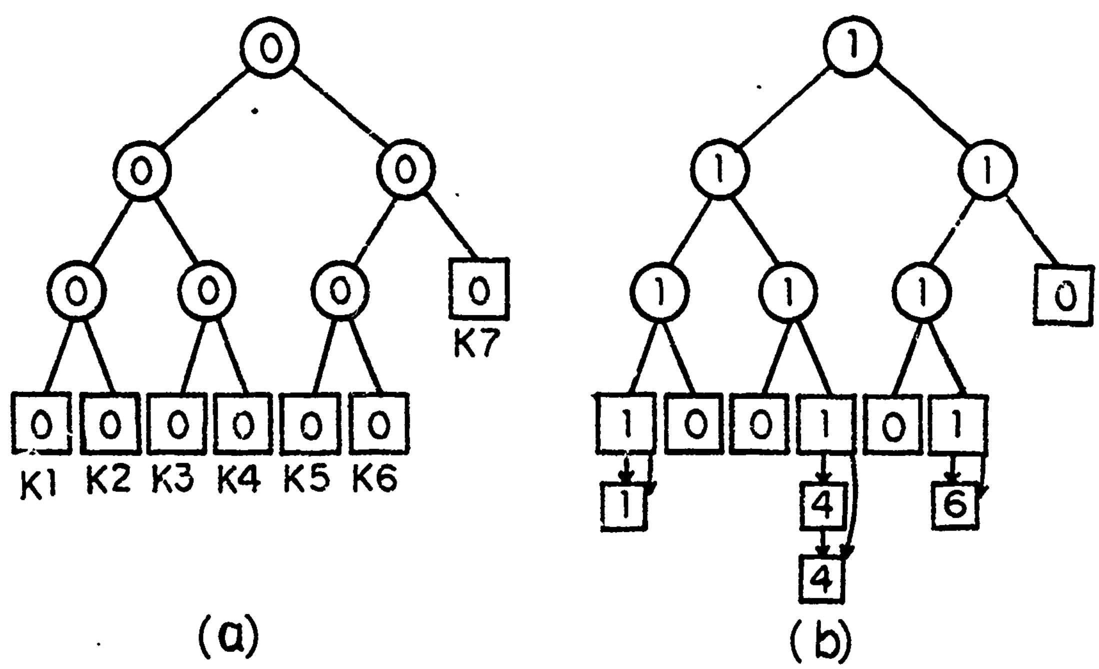


Figure 33

IMPLEMENTATION: In order to represent the empty priority queue it is needed N external nodes, and N-1 internal nodes(as a characteristic of a complete binary tree),total 2N-1 nodes, if priority range is from 1 to N. Internal nodes are implemented as a bit array and have an information bit either 1 or Ø. These nodes are used to find the highest priority item in the queue, during deletion operation.

Before calling the procedure INSERT to put the new item into priority queue, the proper external node is calculated in the main program, such that the priority of the new item will match with the associated key of the external node. The height of the tree will be, $h=\lfloor \log_2(2N-1) \rfloor$ if the number of total nodes are 2N-1 to represent the empty priority queue. Now the proper index K of the external node for the new node can be calculated as follows ; Let I be the index of the rightmost location on the second bottommost level, which will be equal to I=(2**h)-1.

K = I + priority of the new item

IF K > 2N-1 THEN K=(K-(2N-1))+(N-1)

ELSE K=K

Insertion of priority 3, and deletion of highest priority is shown in figure 34 and figure 35 respectivly.

```
PROCEDURE INSERT(K)
/*  K  is  the index of the proper external node of an array
and the priority of the new item is equal to the  associated
key of the external node */
BEGIN
  create a new node y
  IF the external node K is empty  THEN
     link y to K
     set the nodes 1 along path from K to root.
  ELSE
     find the last item belong to external node K, and
     link y to it.
END INSERT.
```

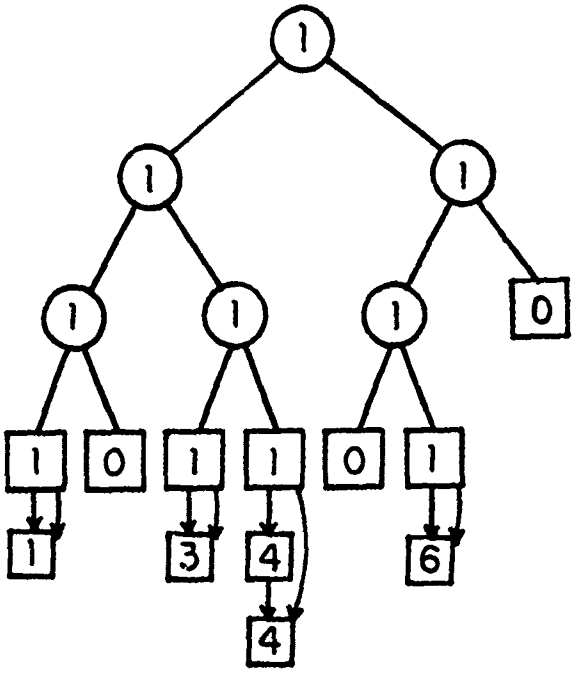


Figure 34. After insertion priority 3 into figure 33.

DELETION: In order to find the highest priority in the tree, the information 1 or 0 at the internal nodes are used. Searching starts at the root, if the right child has information 1, travel is made thru the right child, otherwise thru the left child until it has reached the external nodes. This external node will contain the highest item in the queue. The algorithm for deletion has given below.

```
PROCEDURE DELETE
/* Find the highest priority item and remove it from the
queue. If the removed item is the only one in its catagory,
set nodes (which do not have any relation with other paths)
0 from the external node to the root */
BEGIN
  j=1
  WHILE j < N   DO
    BEGIN
     j=2j
     IF B[2j+1] = 1 THEN  j=j+1
    END
  remove the first item belong to the external node  j
  set the nodes 0 along the path from j to
  the root if necessary.
END DELETE.
```

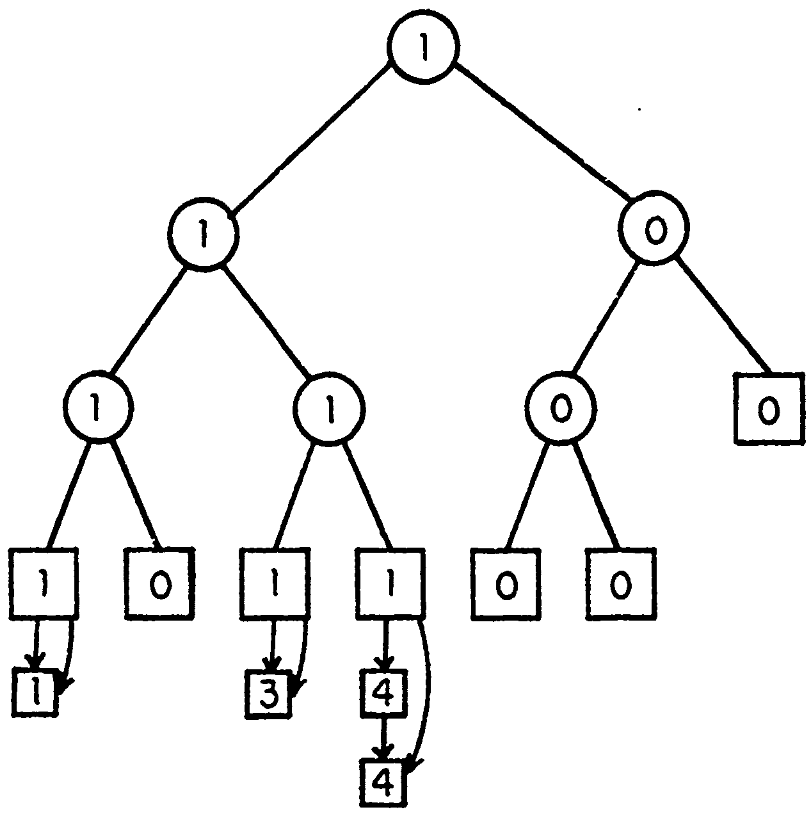


Figure 35. After deletion from figure 34.

FIXED PRIORITY INSERTION WORST CASE ANALYSIS: If the priorities range from 1 to N, in order to construct an empty queue, 2N-1 nodes are needed. The depth of the tree will be $d=\lfloor \log_2(2N-1) \rfloor$. In this method there would not be any key comparisons, one or two addition operations are necessary to find proper position for the new item. This process is done every insertion.

The worst case occurs if the new item is the first item in its priority. After the connection of the new item is done to that external node, it is necessary to traverse along the path from that external node to the root in order to set nodes 1. This would take d steps to reach to the root.

FIXED PRIORITY DELETION WORST CASE ANALYSIS: The worst case occurs if the highest item in the queue is the only one in its category and the path from root to that external node is independent from the other paths in the tree. To find the highest key in the queue takes d steps and after the deletion, travel back thru the root also takes d steps; total 2d steps.

STORAGE REQUIREMENT FOR A FIXED PRIORITY: If priorities range from 1 to n, n external nodes and n-1 internal nodes are needed to construct an empty queue. A bit array of size 2n-1 has to be allocated and in addition to that each external node contains two pointer fields. If there are N

items in the queue (each item contains one pointer field), total required storage would be equal to a bit array of size 2n-1, 2n+N pointer fields, and N*I units space where I is the size of information at each node.

III. AVERAGE CASE TIME ANALYSIS

On a random sequence of inputs, most of these techniques only rarely exhibit the worst case behavior. The running time, especially in the average case is generally more difficult to predict. One method which can give more insight is to determine the expected running time mathematically. Expected running time depends on a probability distribution on the insertion and deletion requests. This approach is called the average analysis of an algorithm[7]. But this kind of analysis turns out to be very difficult for complicated priority queue structures. An alternate method to gain some feeling about the running time of an algorithm is to execute the program several times on "random" inputs and average the results.

This alternate method was used in this research to analyze the algorithms. All programs have been run on the PDP-11 Unix Time Sharing System at NPS. In the empirical test, five different sequence of random numbers which are uniformly distributed between 1 and 1234 were used. Each method (for a specific number of nodes) was run five times by using the same sequence of random numbers and the obtained results were averaged. Tables 2,3,4 give the obtained average running times for each method in seconds. The values in these tables were used to get the graphs which have been given in figures 36, 37, and 38.

The average number of inter-key exchanges during the

insertion process of a 'heap' have been obtained and given in figure 39. Note that the number of inter-key exchanges approach constant value as the number of nodes in the heap approach infinite value.

Finally, an average case behavior and required spaces of an implemented algorithms have been given at table 5. The notation 'O' is called "big-oh" notation and is used in table 5 to express the running times of the algorithms. This notation is a very convenient way for dealing with approximations. In general, the notation O(f(n)) may be used whenever f(n) is a function of a positive integer n; it stands for a quantity which is not explicitly known, except that its magnitude isn't too large. Every appearance of O(f(n)) means precisely this: There is a positive constant M such that the function g(n) represented by O(f(n)) satisfies the condition $g(n) <= M|f(n)|$, for all $n>=n_0$ for some constant n_0.

A. AVERAGE RUNNING TIMES

N-->	100	200	300	400	500	600	700	800	900	1000	1500
HEAP	0.46	0.95	1.43	1.98	2.42	2.97	3.41	3.91	4.44	4.83	7.28
10-ARY	0.33	0.69	1.05	1.4	1.75	2.09	2.46	2.84	3.19	3.56	5.29
LINK LIST	1.75	6.45	14.1	24	37.3	53.2	76.8	97.9	119	----	----
LEFTIST	1,37	3.01	4.7	6.62	8.55	10.5	12.4	14.5	16.5	18.5	28.8
LINK TREE	1.14	2.64	4.3	6.1	7.8	9.84	11.7	13.7	15.6	17.6	29.7
AVL TREE	0.95	2.1	3.32	4.63	5.95	7.28	8.51	99.9	11.5	12.7	19.9
2-3 TREE	1.4	2.95	4.6	6.2	8.1	9.6	11.7	13.4	15.3	17.3	----
FIX PRTY	4.22	8.1	11.7	14.7	17.6	20.3	22.7	25	26.8	28.8	30

Table 2. CPU times in seconds for an 'insertion'.

N-->	100	200	300	400	500	600	700	800	900	1000	1500
HEAP	1.19	2.83	4.07	6.6	8.68	10.8	12.9	15.1	17.7	20.1	31.7
10-ARY	2.1	4.94	6.29	11.5	15.2	18.8	22.3	25.9	29.2	33	52.5
LINK LIST	0.15	0.31	0.56	0.82	0.98	1.48	190	2.10	2.55	----	----
LEFTIST	1.23	2.9	5.2	7.38	10.5	14.2	17.3	20.9	25.7	30.3	57
LINK TREE	0.91	2.76	3.8	5.35	7	8.72	10.4	12.5	14.5	16.2	25.6
AVL TREE	0,61	1.42	2.38	3.29	4.35	5.32	6.51	7.72	8.29	9.04	15.3
2-3 TREE	0.6	1.75	2.9	4	5	6.3	7.36	8.76	9.84	11.1	----
FIX PRTY	.86	1.2	1.5	2.05	2.5	3.0	3.35	3.6	4.3	6.25	8.18

Table 3. CPU times in seconds for a 'deletion'.

N-->	100	200	300	400	500	600	700	800	900	1000	1500
HEAP	1.62	3.78	6.1	8.52	11.1	13.7	16.4	19	22.1	24.9	78.9
10-ARY	2.43	5.63	9.32	12.9	16.9	20.9	24.7	28.7	32.4	36.5	57.7
LINK LIST	1.9	6.75	14.6	24.8	38.3	54.7	72.7	96	121	----	----
LEFTIST	2.6	5.9	9.9	14	19.1	24.7	29.7	35.4	42.2	48.8	65.8
LINK TREE	2.05	5	6.1	11.5	14.6	18.8	22.2	26.2	30.2	33.8	54
AVL TREE	1.56	3.52	5.7	7.92	10.3	12.6	15	17.6	19.6	22.5	75.2
2-3 TREE	2.0	4.7	7.5	10.2	13.1	15.9	19.1	22.2	25.1	28.4	----
FIX PRTY	5.1	9.3	13.2	16.8	20.1	24.5	26.1	28.5	31.1	35.1	44.2

Table 4. CPU times in seconds for 'insertion + deletion'.

B. AVERAGE CASE GRAPHS

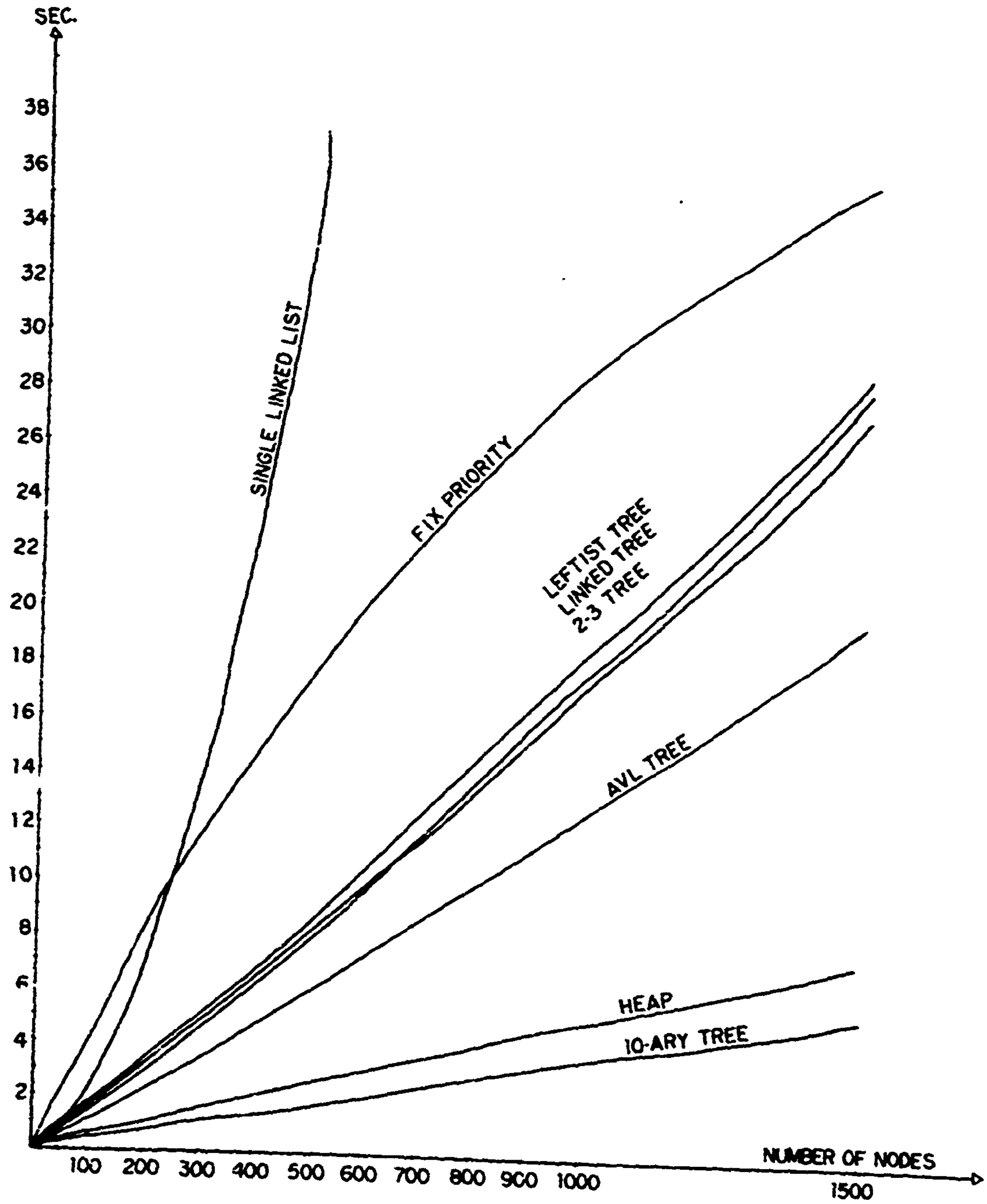


FIGURE 36. RUNNING TIMES FOR INSERTION

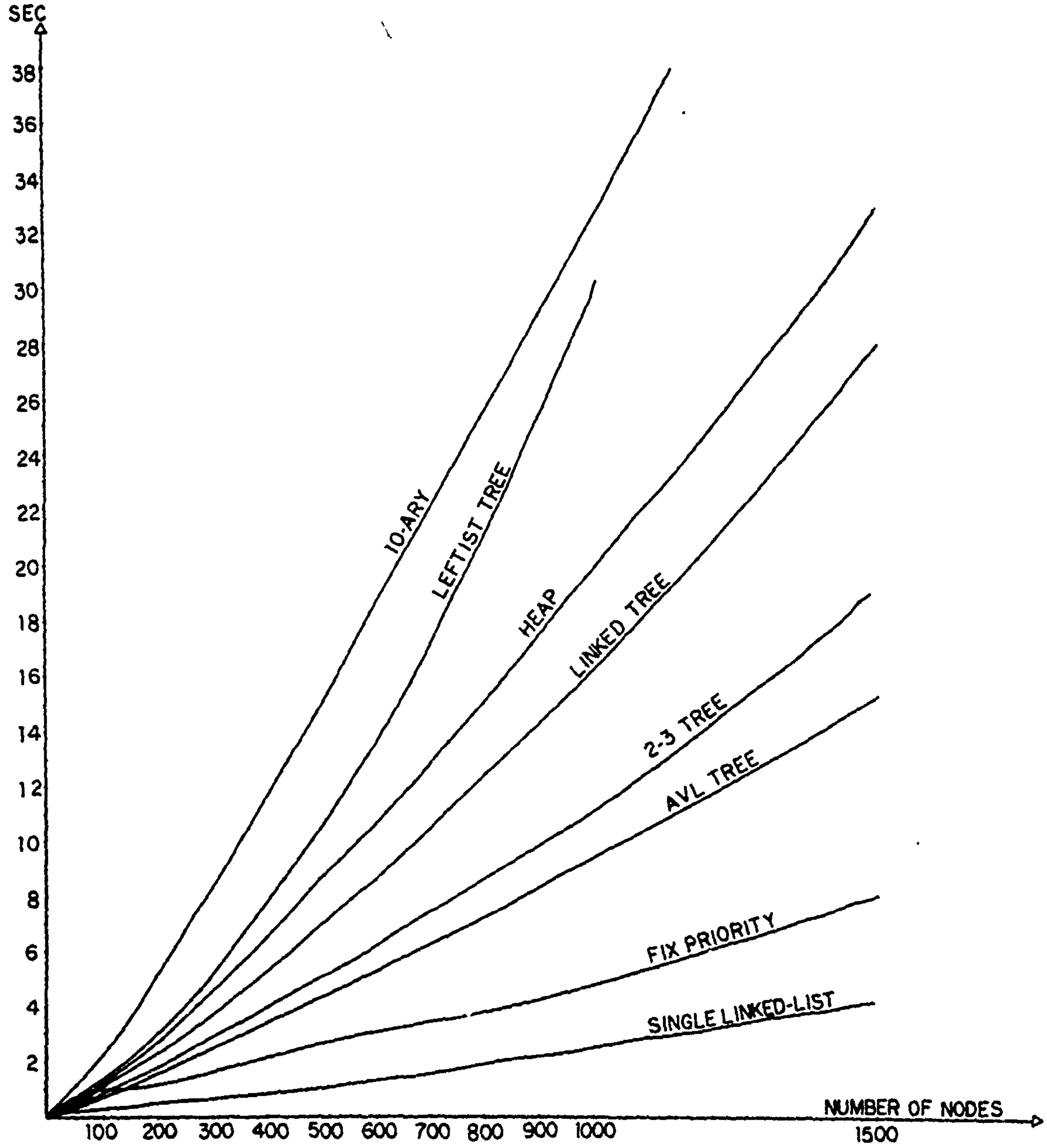


FIGURE 37. RUNNING TIMES FOR DELETION

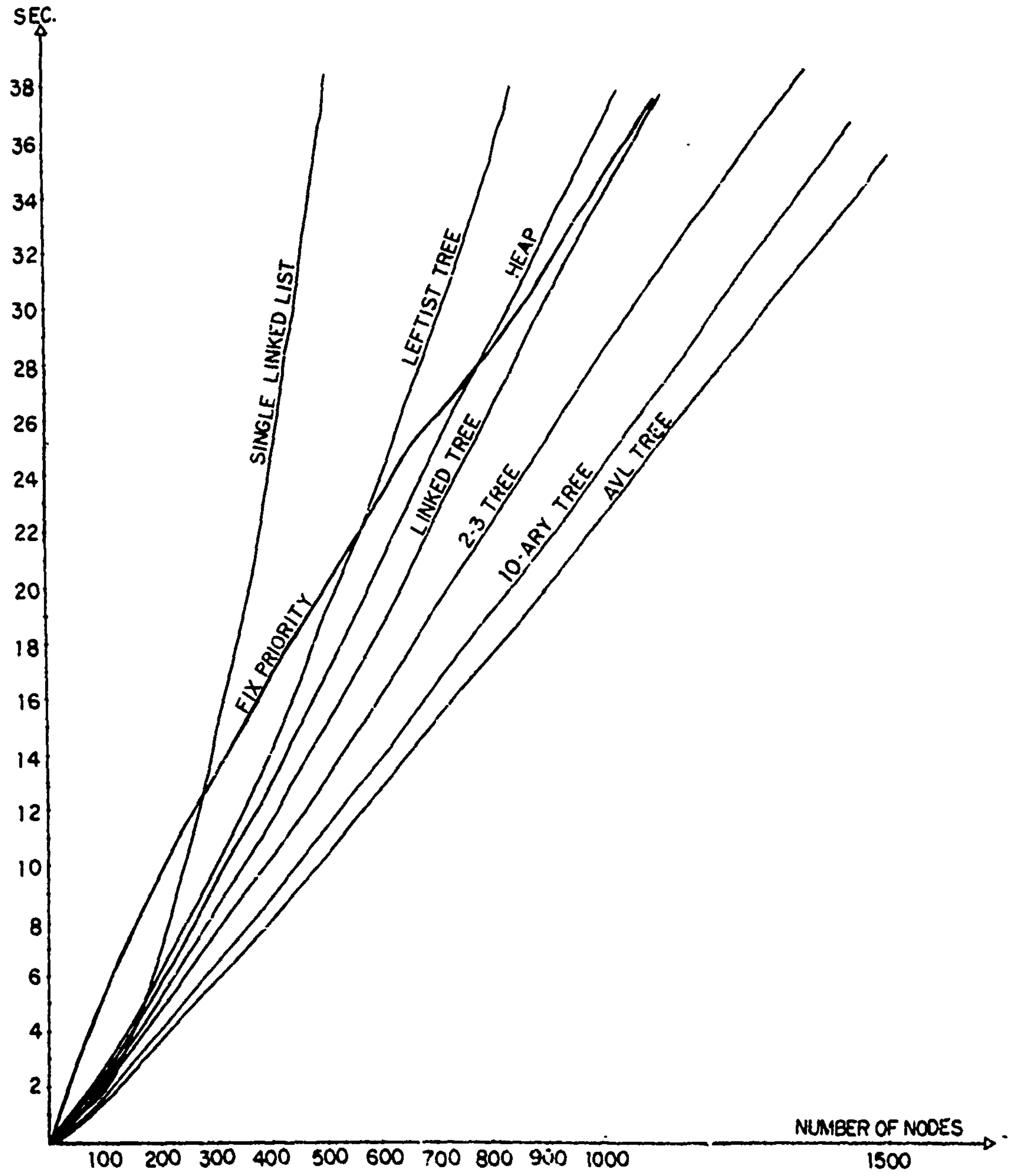


FIGURE 38. RUNNING TIMES FOR INSERTION + DELETION

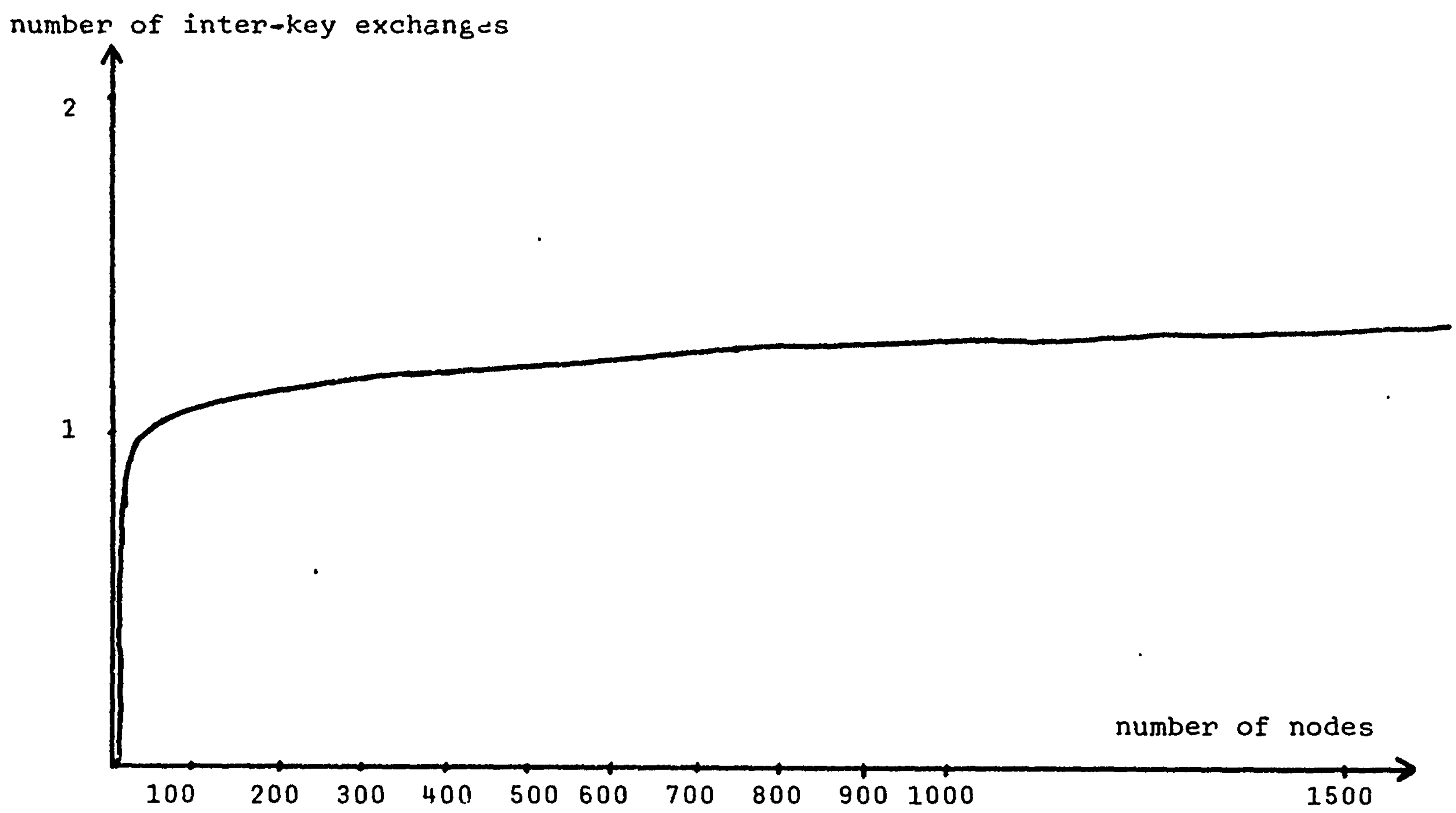

number of inter-key exchanges
2
1
number of nodes
100
200
300
400
500
600
700
800
900
1000
1500

Figure 39.

Priority queue	insertion	deletion	space
heap	O(1)	O(log N)	N(I+1)
k-ary tree	O(1)	O(log N)	N(I-1)
linked-list	O(N)	O(1)	N(I+2)+2
leftist tree	O(log N)	O(log N)	N(I+8)+4
linked tree	O(log N)	O(log N)	N(I+4)
AVL tree	O(log N)	O(log N)	N(I+4)
2-3 tree	O(log N)	O(log N)	N(I+14)-7
fixed prty	O(log n)	O(log n)	4n-1+N(I+1)

Table 5. Conjectured average behavior of an algorithms and required spaces. Where, N is the number of items in the queue, I is the size of information at each node, and n is the priorities range.

Priority queue	insertion	deletion
heap	O(log N)	O(log N)
k-ary tree	O(log N)	O(log N)
linked list	O(N)	O(1)
leftist tree	O(log N)	O(N)
linked tree	O(log N)	O(N)
AVL tree	O(log N)	O(log N)
2-3 tree	O(log N)	O(log N)
fixed prty	O(log n)	O(log n)

Table 6. Summary of the worst case running time of the algorithms where N is the number of items in the queue, and n is the priorities range.

IV. CONCLUSIONS AND RECOMMENDATIONS

When the number of nodes in the priority queue, N, is small, it is best to use one of the straightforward linear list methods to maintain a priority queue; but when N is large such as more than a seventy, a log N method is obviously much faster. Therefore large priority queues are generally represented as heaps or as methods which require O(log N) insertion and deletion time.

Among these algorithms which have been studied, AVL tree structure turned out to be the best in terms of running time on PDP-11 Unix Time Sharing System. In this method, there are neither any inter-key exchanges nor any operations such as multiplication or division which takes more CPU time. But the required space is roughly four times more than heaps and k-ary trees, and programming is more complicated. Heaps and k-ary trees are easy to implement and require minimum space among these algorithms. 2-3 trees also give good running time but required space for 2-3 trees are roughly fourteen times more than heaps. Linked trees require space as much as AVL trees do, but running time is much bigger than AVL trees' running time. Leftist trees are superior for merging disjoint priority queues, but take more space than the AVL trees. If the priorities range is small such as less than fifty, a 'fixed priority' algorithm can be considered to maintain a priority queues efficiently.

As a summary, if in a application there is not any space

constraint, AVL tree structure should be used. If there is not any running time constraint, a heap or k-ary tree structure should be used because heaps and k-ary trees are easy to implement and require minimum storage among these algorithm. Leftist tree structure should be used in a applications which fast merging is required. If the number of nodes is less than hundred, singly linked list could be enough efficient to use.

As an extension of this thesis, a priority queue structure could be implemented by using a 'binomial queues'[4], P-trees[4] and a 'pagoda'. Pagoda is a data structure for representing priority queues and a detailed description can be found in ref. 15. Also dynamic priority queue structures could be studied. A dynamic priority queue is a priority queue with the exception that priorities in the queue can change over time.

APPENDIX. PASCAL CODING OF IMPLEMENTED METHODS.

In this section of the thesis, Pascal coding of the heap, k-ary tree, singly linked list, leftist tree, linked tree, AVL-tree, 2-3 tree and fixed priority have been given respectively. There are not any extra things to do in order to run these programs on the ALTOS system at NPS. In the PDP-11 Unix Time Sharing System there is built in function RANDOM to generate the random numbers, that is why function RANDOM in these coding is not needed on the PDP-11 system. In order to run these programs on the PDP-11 system one needs to set up the function RANDOM in the main program.

A. HEAP

```
(* THIS IS THE IMPLEMENTATION OF A PRIORITY QUEUE BY *)
(* USING A HEAP PROPERTY. DATA TYPE ARRAY IS USED TO *)
(* REPRESENT THE NODES. AN ARRAY A HAS TO BE ALLOCATED*)
(* AS BIG AS THE MAXIMUM SIZE OF THE QUEUE.           *)

PROGRAM HEAP;
CONST  MAX=5000;
       RAN=0.9;
VAR  J :INTEGER;
     SEED:REAL;
     COMD: CHAR;
     A:ARRAY[1..1500] OF INTEGER;
     N,PRTY,EXCH,FIRST,R,P:INTEGER;
     PRT:TEXT;

  FUNCTION RANDOM:INTEGER; (*GENERATES RANDOM NUMBERS BETWEEN *)
    BEGIN                   (* 1 AND 1000 *)
     SEED:=SEED*27.182813+31.415917;
     SEED:=SEED-TRUNC(SEED);
     RANDOM:=1+TRUNC(1000*SEED);
    END;

    PROCEDURE SIFTUP(VAR I:INTEGER);FORWARD;

  PROCEDURE INSERT(VAR PRTY:INTEGER);(*ADDS NEW NODE TO THE QUEUE*)
    BEGIN (* INSERT A NEW NODE INTO A HEAP. *)
     N:=N+1;  R:=N;
     IF N>MAX THEN  WRITELN(PRT,'ERROR')
     ELSE
      BEGIN
       A[N]:=PRTY;
       SIFTUP(R);
      END;
    END;

  PROCEDURE SIFTUP; (*SIFTUP NEWLY INSERTED ITEM *)
   VAR HALF:INTEGER;
       TEMP:INTEGER;
    BEGIN
     WHILE I>1 DO
      BEGIN
       HALF:=I DIV 2;
       IF A[HALF] < A[I] THEN
        BEGIN
         TEMP:=A[I];
         A[I]:=A[HALF];
         A[HALF]:=TEMP;    EXCH:=EXCH+1;
         I:=I DIV 2;
        END
       ELSE  I:=1;
      END;
     END;
```

```
  PROCEDURE SIFTDOWN(VAR I,K:INTEGER);FORWARD;

 PROCEDURE DELETE; (* REMOVES THE HIGHEST PRIORITY IN THE TREE*)
  VAR  TEMP:INTEGER;
   BEGIN
    IF N=Ø THEN  WRITELN(PRT,'ERRØR')
    ELSE
     BEGIN   WHILE N>1 DO  BEGIN
      TEMP:=A[N];
      A[N]:=A[1];
      A[1]:=TEMP;
      N:=N-1; P:=N;  FIRST:=1;
      SIFTDOWN(FIRST,P);
        EXCH:=EXCH+1;
     END;
    END;
   END;


 PROCEDURE BEST; (*RETURNS THE HIGHEST PRIORITY *)
   BEGIN
    IF N<>Ø THEN  WRITE(PRT,A[1])
    ELSE   WRITE(PRT,'NO ELEMENT');
   END;


 PROCEDURE SIFTDOWN;(* SIFTDOWN THE ROOT TO SATISFY HEAP PROPERTY*)
  VAR TEMP:INTEGER;

   BEGIN
    EXCH:=Ø;
    WHILE I <= (K DIV 2) DO
     BEGIN
      IF K=2*I THEN J:=K
      ELSE
        IF A[2*I] > A[2*I+1]  THEN   J:=2*I
        ELSE   J:=2*I+1;
      IF A[I] < A[J]  THEN
       BEGIN
        EXCH:=EXCH+1;
        TEMP:=A[I];
        A[I]:=A[J];
        A[J]:=TEMP;
        I:=J;
       END
      ELSE    I:=(K DIV 2)+1;
     END;
  END;
```

```
BEGIN (*MAIN*)
 REWRITE(PRT,'CONSOLE:');
 SEED:=RAN;
 N:=0;  EXCH:=0;
 WHILE (N <> MAX) DO
  BEGIN
     PRTY:=RANDOM;
     INSERT(PRTY);
  END;
 WRITELN('EXCH= ',EXCH);
 END.
```

B. K-ARY TREE

```
(* THIS IS THE IMPLEMENTATION OF A PRIORITY QUEUE BY *)
(* USING A K-ARY  PROPERTY. AN  ARRAY 'A' IS USED    *)
(* TO REPRESENT NODES IN THE QUEUE.                  *)
PROGRAM KARY;
CONST  MAX=300;
VAR    N,K,PRTY,PIP,P:INTEGER;
       SEED:REAL;
       COMD:CHAR;
       A:ARRAY[1..MAX] OF INTEGER;
       PRT:TEXT;

 FUNCTION RANDOM:INTEGER;(*GENERATES INTEGER RANDOM NUMBERS*)
 BEGIN
  SEED:=SEED*27.182813+31.415917;
  SEED:=SEED-TRUNC(SEED);
  RANDOM:=1+TRUNC(1000*SEED);
 END;

 PROCEDURE BEST;(*FINDS HIGHEST PRIORITY ITEM *)
 BEGIN
  IF  N<>0 THEN  WRITELN(PRT,A[1])
  ELSE  WRITELN(PRT,'NO ITEM IN THE QUEUE'):
 END; (*END OF BEST *)


 PROCEDURE SIFTUP(I:INTEGER);(*SIFTUP THE NEWLY INSERTED NODE*)
 VAR FATHER,TEMP:INTEGER;
 BEGIN
  WHILE  I > 1  DO (* DO IT UNTIL TO GET ROOT. *)
   BEGIN
   FATHER:=(I+K-2) DIV K;    (* FATHER OF THE NEW ITEM. *)
   IF  A[FATHER] < A[I]  THEN (*SIFTUP NEW ITEM.*)
    BEGIN           (* EXCHANGE FATHER AND SON *)
    TEMP:=A[I];
    A[I]:=A[FATHER];
    A[FATHER]:=TEMP:
    I:=FATHER;
    END
   ELSE I:=1:(*NEW ITEM IN PROPER POSITION,LEAVE IT THERE.*)
   END;
  END;   (* END OF SIFTUP *)


 PROCEDURE INSERT( PRTY:INTEGER);
 BEGIN  (* ADD A NEW NODE INTO A TREE *)
  N:=N+1;
  IF  N >= MAX THEN  WRITELN(PRT,'ERROR') (* QUEUE IS FULL. *)
  ELSE       (* INSERT NEW ITEM IN Nth POSITION.*)
   BEGIN
   A[N]:=PRTY:
   A[N+1]:=0:  (*ZERO AT Nth POSITION IS TERMINATE SYMBOL.*)
   SIFTUP(N);  (* MOVE NEW ITEM THRU THE ROOT. *)
   END;
 END;  (* END OF INSERT *)
```

```
PROCEDURE SIFTDOWN( L,Z:INTEGER); (* SIFTDOWN THE ROOT TO *)
VAR COUNT,FIRST,J,TEMP:INTEGER;   (* SATISFY K-ARY PROPERTY*)
BEGIN
 WHILE  L <= (Z+K-2) DIV K  DO (*DO IT UNTIL LOWEST LEVEL*)
  BEGIN
   FIRST:=(K*L)-(K-2); (* THE FIRST SON OF FATHER FROM LEFT*)
   J:=FIRST+1;         (* THE SECOND SON OF FATHER FROM LEFT*)
   COUNT:=1;
   WHILE  (COUNT < K) AND  (A[J] <> 0 )  DO (*DO IT UNTIL TO*)
    BEGIN (*GET TERMINATE SYMBOL OR RIGHT MOST SON OF FATHER*)
    IF  A[FIRST] > A[J]  THEN  (* FIND LARGEST SON *)
     BEGIN
     J:=J+1;
     COUNT:=COUNT+1;
     END
    ELSE
     BEGIN
     FIRST:=J;
     J:=J+1;
     COUNT:=COUNT+1;
     END;
    END;

 IF  A[L] < A[FIRST]  THEN
   BEGIN         (* EXCHANGE LARGEST SON AND FATHER. *)
    TEMP:=A[L];
    A[L]:=A[FIRST];
    A[FIRST]:=TEMP;
    L:=FIRST;
   END
 ELSE
   L:=((Z+K-2) DIV K)+1; (*THE ITEM IS IN PROPER PLACE*)
 END;
END; (* END OF PROCEDURE SIFTDOWN *)
```

```
PROCEDURE DELETE; (* REMOVE THE HIGHEST PRIORITY *)
BEGIN
 IF N=0 THEN  WRITELN(PAT,'NO ITEM TO DELETE')
 ELSE
  BEGIN
  A[N+1]:=A[1];(*MOVE HIGHEST PRIORITY ITEM TO N+1th POSITION*)
  A[1]:=A[N];   (*MOVE LAST ITEM IN QUEUE TO FIRST POSITION.*)
  A[N]:=0;      (* ZERO TO INDICATE TERMINATE SYMBOL.*)
  N:=N-1;  P:=N;  FIR:=1;
  SIFTDOWN(FIR,P);(*SIFTDOWN THE FIRST ITEM TO PROPER POSITION*)
  END;
 END;  (* END OF PROCEDURE DELETE.*)
```

```
PROCEDURE PRINT;
 VAR NUM:INTEGER;
BEGIN
 NUM:=1;  WRITELN(PRT,'..N=',N);
 WHILE NUM <=  N+1  DO
  BEGIN
  WRITE(PRT,A[NUM]);
  WRITE(PRT,' ');
  NUM:=NUM+1;
  END;
END;



BEGIN (*MAIN*)
REWRITE(PRT,'CONSOLE:');
SEED:=0.2000;
N:=0;
WRITE(PRT,'WHAT IS THE DEGREE OF THREE..?  K: ');
READLN(K);  WRITE(PRT,'     ',K);
WHILE N < MAX-1 DO
 BEGIN
 WRITE(PRT,'>');
 READLN(COMD);
 IF COMD='I' THEN       (* COMMAND FOR INSERTION *)
  BEGIN
  PRTY:=RANDOM;
  WRITELN(PRT,'RANDOM= ',PRTY);
  INSERT(PRTY);
  END
 ELSE
  IF  COMD='D' THEN   DELETE  (* COMMAND FOR DELETION *)
  ELSE  BEST;  (* FIND THE HIGHEST PRIORITY IN THE QUEUE*)
   PRINT;
 END;
END.
```

C. SINGLE LINKED-LIST

```
(* THIS THE IMPLEMENTATION OF A PRIORITY QUEUE BY     *)
(* USING A SINGLY LINKED LIST PROPERTY. A DATA TYPE   *)
(* RECORD IS USED TO REPRESENT THE NODES IN THE QUEUE *)

PROGRAM SINGLELINK;
CONST  MAX=300;
TYPE PTR=^NODE;
     NODE=RECORD
          LINK:PTR;
          KEY:INTEGER;
          END;
VAR FRONT,BACK,N:PTR;
    NUM:INTEGER;
    CCMD:CHAR;
    SEED:REAL;
    PRT:TEXT;


FUNCTION RANDOM:INTEGER; (*GENERATES RANDOM NUMBER *)
BEGIN                    (* BETWEEN 1 AND 1000*)
 SEED:=SEED * 27.182813 + 31.415917;
 SEED:=SEED-TRUNC(SEED);
 RANDOM:=1+TRUNC(1000*SEED);
END;


PROCEDURE DELETE; (* REMOVES THE NODE WITH HIGHEST PRTY.*)
VAR HIGH:INTEGER;
BEGIN
  IF NUM = 0  THEN
     WRITELN(PRT,'THERE IS NO ITEM IN THE QUEUE',
  ELSE
     IF NUM=1 THEN(*THERE IS ONLY ONE ITEM IN THE QUEUE.*)
       BEGIN
        HIGH:=FRONT^.KEY;
        FRONT:=NIL;
        BACK:=NIL;
       END
     ELSE (*THERE ARE MORE THAN ONE ITEM IN THE QUEUE.*)
       BEGIN
        HIGH:=FRONT^.KEY;
        FRONT:=FRONT^.LINK;
       END;
 END:  (* END OF PROCEDURE DELETE.*)


 PROCEDURE BEST;
 BEGIN
   IF NUM = 0 THEN
    WRITELN(PRT,'THERE IS NO ITEM IN THE QUEUE.')
   ELSE
    WRITELN(PRT,'HIGHEST PRIORITY IS: ',FRONT^.KEY):
 END:  (* END OF PROCEDURE BEST.*)
```

```
PROCEDURE INSERT; (* ADDS THE NEW NODE TO THE QUEUE *)
VAR W:PTR;
BEGIN
 IF NUM = 1 THEN  (* FIRST ITEM CAME IN THE QUEUE.*)
  BEGIN
    NEW(N);         (* CREATE NEW NODE AND INITIALIZE *)
    N^.KEY:=RANDOM;
    FRONT:=N;
    BACK:=N;
    N^.LINK:=NIL;
  END
 ELSE (* THERE IS AT LEAST ONE ITEM IN THE QUEUE.*)
  BEGIN
    NEW(N);
    N^.KEY:=RANDOM;    WRITELN(PRT,'RANDOM: ',N^.KEY';
    N^.LINK:=NIL;
    W:=FRONT;
    IF W^.KEY < N^.KEY  THEN (*HIGHEST PRIORITY CAME IN*)
     BEGIN
      N^.LINK:=FRONT;
      FRONT:=N;
     END
    ELSE
      IF W^.LINK = NIL THEN (* THERE IS ONLY ONE ITEM *)
       BEGIN
         W^.LINK:=N;
         BACK:=N;
       END
      ELSE (*THERE ARE AT LEAST TWO ITEMS IN THE QUEUE*)
      BEGIN
       WHILE (W^.LINK^.KEY >= N^.KEY) AND (W^.LINK <> BACK) DO
         W:=W^.LINK; (* FIND PROPER PLACE FOR NEW ITEM.*)
       IF W^.LINK^.KEY < N^.KEY  THEN
        BEGIN
         N^.LINK:=W^.LINK;
         W^.LINK:=N;
        END
       ELSE    (* INSERT NEW ITEM AS AN LAST ITEM.*)
        BEGIN
         W^.LINK^.LINK:=N;
         BACK:=N;
        END;
      END;
     END;
  END;
```

```
BEGIN    (*MAIN PROGRAM*)
REWRITE(PRT,'CONSOLE:',;
SEED:=0.20000;
NUM:=0;
FRONT:=NIL;
BACK:=NIL;
WHILE NUM < MAX DO
 BEGIN
 WRITE(PRT,'>');
 READLN(COMD);
 IF COMD = 'I' THEN (* COMMAND FOR INSERTION.*)
  BEGIN
   NUM:=NUM+1;
   INSERT;
  END
 ELSE
  IF COMD = 'D' THEN (*COMMAND FOR DELETION.*)
   BEGIN
   NUM:=NUM-1;
   DELETE;
   END
  ELSE  BEST;  (* FIND THE HIGHEST ITEM IN THE QUEUE.*)
   PRINT;   (* DISPLAY PRIORITIES IN THE QUEUE.*)
 END;
END.  (* END OF MAIN PROGRAM *)
```

D. LEFTIST TREE

```
(* THIS IS THE IMPLEMENTATION OF A PRIORITY QUEUE BY USING*)
(* A LEFTIST TREE PROPERTY. RECORD IS USED TO REPRESENT NODES*)
PROGRAM LEFTIST;
CONST MAX=100;
TYPE PTR=^NODE;
     NODE=RECORD
          LEFT,RIGHT:PTR;
          KEY,DIST:INTEGER;
          END;
VAR ROOT:PTR;
    NUM,PRTY:INTEGER;
    COMD:CHAR;
    SEED:REAL;
    H:BOOLEAN;

PROCEDURE INSERT(PRTY:INTEGER; VAR P:PTR; VAR H:BOOLEAN);
VAR N:PTR;   TEMP,I:INTEGER;
BEGIN  (* INSERT THE NEW NODE INTO A LEFTIST TREE.*)
 IF P^.DIST = 0 THEN  (* IS IT LEAF NODE ?*)
 BEGIN
   P^.KEY:=PRTY;
   P^.DIST:=1;
   H:=TRUE;
   FOR I:=1 TO 2 DO
   BEGIN        (*CREATE 2 EMPTY NODES FOR LEAF NODES*)
    NEW(N);
    N^.DIST:=0;
    N^.KEY:=0;
    N^.LEFT:=NIL;
    N^.RIGHT:=NIL;
    IF I = 1 THEN  P^.LEFT:=N
    ELSE          P^.RIGHT:=N;
   END;
 END
 ELSE
   IF P^.KEY >= PRTY THEN
   BEGIN      (* ROOT'S KEY IS BIGGER THAN NEW ITEM'S PRTY*)
    IF P^.LEFT^.DIST <= P^.RIGHT^.DIST  THEN
     BEGIN    (* GO THRU LEFT BRANCH *)
     INSERT(PRTY,P^.LEFT,H);
     H:=FALSE;  (* INSERTION THRU LEFT DOES NOT GROW THE HEIGHT*)
     END
    ELSE
     BEGIN      (* GO THRU RIGHT BRANCH *)
     INSERT(PRTY,P^.RIGHT,H);
     IF  H  THEN  P^.DIST:=P^.DIST + 1; (* INCREMENT HEIGHT*)
     END;
   END
   ELSE    (*NEW ITEM IS BIGGER THAN ROOT'S KEY *)
   BEGIN   (*EXCHANGE KEYS *)
    TEMP:=P^.KEY;
    P^.KEY:=PRTY;
    PRTY:=TEMP;
    INSERT(PRTY,P,H);
   END;
 END;
```

```
PROCEDURE MERGE(VAR P1,P2:PTR);(*AFTER DELETION OF THE*)
VAR P3:PTR;              (*ROOT MERGES ITS TWO SUBTREES *)
BEGIN
 IF P2^.DIST = 0 THEN  P2:=P1
 ELSE
   IF P1^.KEY > P2^.KEY THEN
    BEGIN
    P3:=P2;
    P2:=P1;
    P1:=P3;
    MERGE(P1,P2^.LEFT);
    END
   ELSE  MERGE(P1,P2^.LEFT);
   IF ROOT^.LEFT^.DIST < ROOT^.RIGHT^.DIST  THEN
    BEGIN  (*EXCHANGE LEFT AND RIGHT SUBTREES*)
    P3:=ROOT^.LEFT;
    ROOT^.LEFT:=ROOT^.RIGHT;
    ROOT^.RIGHT:=P3;
    END;
END;


PROCEDURE DELETE( P:PTR); (* REMOVES THE HIGHEST ITEM.*)
BEGIN
 IF  NUM = 1 THEN
   BEGIN
   ROOT^.DIST:=0;
   ROOT^.KEY:=0;
   ROOT^.LEFT:=NIL;
   ROOT^.RIGHT:=NIL;
   END
  ELSE
   IF NUM = 0 THEN  WRITELN('NO ITEM IN THE QUEUE')
   ELSE
 IF P^.LEFT^.KEY > P^.RIGHT^.KEY  THEN
  BEGIN     (*MAKE LEFT SON ROOT AND MERGE *)
   ROOT:=P^.LEFT;  (* LEFT AND RIGHT SUBTREES*)
   MERGE(P^.RIGHT,P^.LEFT^.LEFT);
  END
 ELSE
  BEGIN     (*MAKE RIGHT SON ROOT AND MERGE *)
   ROOT:=P^.RIGHT;    (* LEFT AND RIGHT SUBTREES*)
   MERGE(P^.LEFT,P^.RIGHT^.LEFT);
  END;
END;

FUNCTION RANDOM:INTEGER; (*GENERATES RANDOM NUMBERS*)
BEGIN
SEED:= SEED * 27.182813 + 31.415917;
SEED:=SEED - TRUNC(SEED);
RANDOM:=1 + TRUNC(MAX * SEED);
END;
```

```
PROCEDURE PRINT(TEST:PTR);
BEGIN
 IF TEST^.KEY <> 0 THEN
  BEGIN
  WRITE(' ');
  WRITE(TEST^.KEY);
  PRINT(TEST^.LEFT);
  PRINT(TEST^.RIGHT);
  END
 ELSE WRITE('=');
END;


BEGIN (* MAIN *)
NUM:=0;
SEED:=0.2000;
H:=FALSE;
NEW(ROOT);
ROOT^.DIST:=0;
ROOT^.KEY:=0;
ROOT^.LEFT:=NIL;
ROOT^.RIGHT:=NIL;
WHILE NUM < MAX DO
 BEGIN
 WRITE('>');
 READLN(COMD);
 IF COMD = 'I' THEN
  BEGIN
  NUM:=NUM + 1;
  H:=FALSE;
  PRTY:=RANDOM;
  WRITELN('RANDOM:',PRTY);
  INSERT(PRTY,ROOT,H);
  END
 ELSE
  IF COMD = 'D' THEN
  BEGIN
  IF NUM = 0 THEN WRITELN('THERE IS NO ITEM')
  ELSE
   BEGIN
   NUM:=NUM - 1;
   DELETE(ROOT);
   END;
  END;
  PRINT(ROOT);
 END;
END.
```

E. LINKED TREE

```
(* THIS IS THE IMPLEMENTATION OF A PRIORITY QUEUE BY *)
(* USING A LINKED-TREE PROPERTY. A DATA TYPE RECORD IS *)
(* USED TO REPRESENT NODES..                            *)

PROGRAM LINKEDTREE;
TYPE PTR=^NODE;
     NODE=RECORD
          LEFT,RIGHT:PTR;
          KEY,DESC:INTEGER;
          END;
VAR NUM:INTEGER;
    SEED:REAL;
    PRT:TEXT;
    N,V,ROOT:PTR;
    COMD:CHAR;

 PROCEDURE INSERT(W:PTR ; PRTY:INTEGER);(*ADDS NEW NODE*)
 VAR TEMP:INTEGER;
 BEGIN (* INSERTS NEW NODE INTO THE TREE. *)
  IF NUM=1 THEN ROOT:=N (*FIRST ITEM *)
  ELSE
   BEGIN
   IF W^.KEY >=  PRTY  THEN (*NEW ITEM IS SMALLER *)
     BEGIN
       IF W^.LEFT <> NIL THEN (* W HAS LEFT SON *)
         BEGIN
           IF W^.RIGHT <> NIL THEN (* W ALSO HAS RIGHT SON *)
             BEGIN
               IF W^.LEFT^.DESC >= W^.RIGHT^.DESC  THEN
                 BEGIN  (* TRAVEL THRU RIGHT BRANCH *)
                  W:=W^.RIGHT;
                  W^.DESC:=W^.DESC+1;
                  INSERT(W,N^.KEY);
                 END
               ELSE
                 BEGIN (* TRAVEL THRU LEFT BRANCH *)
                  W:=W^.LEFT;
                  W^.DESC:=W^.DESC+1;
                  INSERT(W,N^.KEY);
                 END;
              END
            ELSE  W^.RIGHT:=N; (* LINK NEW ITEM AS RIGHT SON*)
         END
        ELSE  W^.LEFT:=N;(*LINK NEW ITEM AS LEFT SON*)
     END
   ELSE
     BEGIN (* KEY EXCHANGES ARE NECESSARY *)
      TEMP:=W^.KEY;
      W^.KEY:=N^.KEY;
      N^.KEY:=TEMP;
      INSERT(W,N^.KEY);
     END;
 END;
 END; (* END OF INSERT *)
```

```
PROCEDURE DELETE(VAR X:PTR); (* REMOVES THE HIGHEST ITEM *)
VAR Y,Z:PTR;
BEGIN
  Y:=X^.LEFT;
  Z:=X^.RIGHT;
  IF Y  <> NIL  THEN (* LEFT SUBTREE EXIST *)
   BEGIN
   IF Z  <> NIL   THEN (* RIGHT SUBTREE EXIST *)
    BEGIN
     IF Y^.KEY >= Z^.KEY  THEN
      BEGIN (* MOVE LEFT SON TO THE PARENT POSITION*)
       X^.KEY:= Y^.KEY;
       Y^.DESC:=Y^.DESC - 1;
       IF Y^.DESC < 0  THEN  X^.LEFT:=NIL(*REACHED TO LEAF*)
       ELSE  DELETE(X^.LEFT);
      END
     ELSE
      BEGIN (* MOVE RIGHT SON TO THE ITS PARENT POSITION*)
       X^.KEY:=Z^.KEY;
       Z^.DESC:=Z^.DESC-1;
       IF Z^.DESC < 0  THEN  X^.RIGHT:=NIL(*REACHED TO LEAF*)
       ELSE   DELETE(X^.RIGHT);
      END;
    END
   ELSE   X:=X^.LEFT;(*RIGHT SUBTREE DOES NOT EXIST *)
   END
  ELSE  X:=X^.RIGHT; (*LEFT SUBTREE DOES NOT EXIST*)
END;


PROCEDURE BEST(VAR TEST:PTR);
BEGIN (*RETURNS THE NODE WITH HIGHEST PRTY*)
  IF TEST=NIL THEN  WRITELN(PRT,'NO ITEM IN QUEUE.')
  ELSE  WRITELN(PRT,'HIGHEST :',TEST^.KEY);
END;

FUNCTION RANDOM:INTEGER;(*GENERATES RANDOM NUMBERS*)
 BEGIN
  SEED:=SEED * 27.182813 + 31.415917;
  SEED:=SEED - TRUNC(SEED);
  RANDOM:=1 + TRUNC(100*SEED);
 END;


PROCEDURE PRINT (TEST:PTR);
BEGIN
 IF TEST <> NIL THEN
  BEGIN
  WRITE(PRT,'  ');
  WRITE(PRT,TEST^.KEY);
  PRINT(TEST^.LEFT);
  PRINT(TEST^.RIGHT);
  END
 ELSE WRITE(PRT,'=');
END;
```

```
BEGIN   (*MAIN PROGRAM *)
 REWRITE(PRT,'CONSOLE:');
 SEED:=0.2200;
 NUM:= 0;
 ROOT:=NIL;
 WHILE NUM < 100 DO
 BEGIN
 WRITE(PRT,'>');
 READLN(COMD);
 IF COMD='I' THEN
  BEGIN (* CREATE NEW NODE AND INITIALIZE *)
  NEW(N);
  N^.KEY:=RANDOM;
  N^.LEFT:=NIL;
  N^.RIGHT:=NIL;
  N^.DESC:=0;
  NUM:=NUM+1;
  INSERT(ROOT,N^.KEY);
  PRINT(ROOT);
  END
ELSE
IF COMD = 'D' THEN
   BEGIN
    NUM:=NUM-1;
     IF NUM < 0  THEN  WRITELN('NO ITEM IN THE QUEUE')
     ELSE
      IF NUM = 0 THEN ROOT:=NIL (*LAST ITEM IS DELETED*)
      ELSE
       IF ROOT^.LEFT = NIL  THEN  ROOT:=ROOT^.RIGHT
       ELSE
         IF ROOT^.RIGHT = NIL  THEN  ROOT:=ROOT^.LEFT
         ELSE
          BEGIN
            V:=ROOT;
            DELETE(V);
          END
     END
  ELSE
    IF COMD = 'B' THEN  BEST(ROOT)
        ELSE  WRITELN(PRT,'INVALID COMMAND');
  END;
  END.
```

F. AVL-TREE

```
(* THIS IS THE IMPLEMENTATION OF A PRIORITY QUEUE BY *)
(* USING AN AVL-TREE PROPERTY. A DATA TYPE RECORD IS *)
(* USED TO REPRESENT NODES.                          *)

PROGRAM AVLTREE;
TYPE PTR=^NODE;
     NODE=RECORD
          KEY:INTEGER;
          LEFT,RIGHT:PTR;
          BAL:-1..+1;
          END;
VAR ROOT:PTR;
    H:BOOLEAN;
    NUM,PRTY:INTEGER;
    SEED:REAL;
    COMD:CHAR;

PROCEDURE INSERT(X:INTEGER; VAR P:PTR; VAR H:BOOLEAN);
VAR P1,P2:PTR;
BEGIN (* INSERTS THE NEW NODE INTO TREE. *)
  IF P = NIL THEN (*IS REACHED LEAF NODE,INSERT NEW ITEM*)
  BEGIN    (*CREATE NEW NODE AND INITIALIZE. *)
   NEW(P);
   H:=TRUE;   (* SUBTREE HEIGHT IS INCREASED *)
   WITH P^ DO
    BEGIN
     KEY:=X;
     LEFT:=NIL;
     RIGHT:=NIL;
     BAL:=0;
    END;
  END
  ELSE
    IF X < P^.KEY THEN (*NEW ITEM IS LESS THAN ROOT P*)
     BEGIN
      INSERT(X,P^.LEFT,H); (* GO THRU LEFT SON *)
      IF H THEN  (*LEFT BRANCH HAS GROWN HIGHER *)
      BEGIN
      CASE P^.BAL  OF
   0: P^.BAL:=-1;   (*THE WEIGHT IS SLANTED TO THE LEFT*)
   1: BEGIN         (*THE PREVIOUS IMBALANCE AT P HAS  *)
      P^.BAL:=0;       (*  BEEN EQUILIBRATED. *)
      H:=FALSE;
      END;
  -1: BEGIN         (* REBALANCE SUBTREE. *)
      P1:=P^.LEFT;
      IF P1^.BAL = -1  THEN
       BEGIN          (* DO LL ROTATION *)
        P^.LEFT:=P1^.RIGHT;
        P1^.RIGHT:=P;
        P^.BAL:=0;
        P:=P1;
       END
      ELSE
```

```
      BEGIN         (* DO LR  ROTATION *)
       P2:=P1^.RIGHT;
       P1^.RIGHT:=P2^.LEFT;
       P2^.LEFT:=P1;
       P^.LEFT:=P2^.RIGHT;
       P2^.RIGHT:=P;
        IF P2^.BAL = -1  THEN  P^.BAL:=+1
        ELSE  P^.BAL:=0;
        IF P2^.BAL = +1  THEN  P1^.BAL:=-1
        ELSE  P1^.BAL:=0;
       P:=P2;
       END;
     P^.BAL:=0;
     H:=FALSE;
    END;
   END;  END      ELSE WRITELN(' ');
 END
 ELSE
   BEGIN
     INSERT(X,P^.RIGHT,H);  (* GO THRU RIGHT SON *)
     IF H THEN   (* RIGHT BRANCH HAS GROWN HIGHER *)
     BEGIN
     CASE P^.BAL OF
    0:P^.BAL:=+1; (*THE WEIGHT IS SLANTED TO THE RIGHT*)
    -1:BEGIN       (*THE PREVIOUS INBALANCE AT P HAS   *)
       P^.BAL:=0;      (* BEEN EQUILIBRATED *)
       H:=FALSE;
       END;
     1:BEGIN           (* REBALANCE SUBTREE *)
       P1:=P^.RIGHT;
       IF P1^.BAL = +1  THEN
         BEGIN      (* DO  RR  ROTATION *)
         P^.RIGHT:=P1^.LEFT;
         P1^.LEFT:=P;
         P^.BAL:=0;
         P:=P1;
         END
       ELSE
         BEGIN      (* DO  RL  ROTATION *)
          P2:=P1^.LEFT;
          P1^.LEFT:=P2^.RIGHT;
          P2^.RIGHT:=P1;
          P^.RIGHT:=P2^.LEFT;
          P2^.LEFT:=P;
           IF P2^.BAL = +1  THEN  P^.BAL:= -1
           ELSE  P^.BAL:=0;
           IF P2^.BAL = -1  THEN  P1^.BAL:=+1
           ELSE  P1^.BAL:=0;
          P:=P2;
          END;
       P^.BAL:=0;
       H:=FALSE;
      END;
     END;  END
    ELSE WRITELN(' ');  END;
    END;
```

```
FUNCTION RANDOM:INTEGER;
BEGIN
 SEED:=SEED * 27.182813 + 31.415917;
 SEED:=SEED - TRUNC(SEED);
 RANDOM:=1 + TRUNC(100*SEED);
END;


 PROCEDURE BALANC(VAR P:PTR; VAR H:BOOLEAN);
 VAR P1,P2:PTR; (* REBALANCE THE TREE AFTER DELETION*)
     BAL1,BAL2:-1..+1;
 BEGIN    (* H=TRUE,RIGHT BRANCH HAS BECOME LESS HIGHER*)
   CASE P^.BAL OF
 1:P^.BAL:=0;
 0:BEGIN
   P^.BAL:=-1;
   H:=FALSE;
   END;
-1:BEGIN          (* REBALANCE SUBTREE *)
   P1:=P^.LEFT;
   BAL1:=P1^.BAL;
   IF BAL1 <= 0  THEN
   BEGIN          (* DO  LL  ROTATION *)
    P^.LEFT:=P1^.RIGHT;
    P1^.RIGHT:=P;
    IF BAL1 = 0  THEN
      BEGIN
       P^.BAL:=-1;
       P1^.BAL:=+1;
       H:=FALSE;
      END
    ELSE
      BEGIN
       P^.BAL:=0;
       P1^.BAL:=0;
      END;
    P:=P1;
    END
    ELSE
     BEGIN         (* DO  LR  ROTATION  *)
       P2:=P1^.RIGHT;
       BAL2:=P2^.BAL;
       P1^.RIGHT:=P2^.LEFT;
       P2^.LEFT:=P1;
       P^.LEFT:=P2^.RIGHT;
       P2^.RIGHT:=P;
       IF BAL2 = -1  THEN  P^.BAL:=+1
       ELSE  P^.BAL:=0;
       IF BAL2 = +1  THEN  P1^.BAL:=-1
       ELSE  P1^.BAL:=0;
     P:=P2;
     P2^.BAL:=0;
     END;
   END;
  END;
END;
```

```
PROCEDURE DELETE(VAR P:PTR; VAR H:BOOLEAN);
VAR Q:PTR;
BEGIN (* DELETES THE NODE WITH HIGHEST PRIORITY *)
  IF NUM = 0 THEN   (* QUEUE IS EMPTY *)
    BEGIN
     WRITELN('QUEUE IS EMPTY');
     H:=FALSE;
    END
  ELSE
   BEGIN
    IF P^.RIGHT <> NIL THEN
    BEGIN (* SEARCH UNTIL TO REACH LEAF NODE *)
     DELETE(P^.RIGHT,H); (* GO THRU RIGHT SON *)
     IF H THEN  BALANC(P,H); (* REBALANCE SUBTREE*)
    END
    ELSE
     BEGIN
     Q:=P;
     P:=Q^.LEFT;
     H:=TRUE;(*HEIGHT OF THE SUBTREE HAS BEEN REDUCED*)
     END;
    END;
 END;


 BEGIN
 NUM:=0;
 SEED:=0.2000;
 H:=FALSE;
 ROOT:=NIL;
 WHILE NUM < 100 DO
 BEGIN
  WRITE('>');
  READLN(COMD);
  IF COMD = 'I'  THEN
    BEGIN
      NUM:=NUM+1;
      PRTY:=RANDOM;
      H:=FALSE;
      INSERT(PRTY,ROOT,H);
    END
    ELSE
      IF  COMD ='D'  THEN
       BEGIN
        H:=FALSE;
        DELETE(ROOT,H);
        NUM:=NUM-1;
       END
      ELSE WRITELN('INVALID COMMAND');
  END;
  END.
```

G. 2-3 TREE

```
(*THIS IS THE IMPLEMENTATION OF A PRIORITY QUEUE BY *)
(*USING A 2-3 TREE PROPERTY. A DATA TYPE RECORD IS  *)
(* USED TO REPRESENT THE NODES IN THE TREE.         *)
PROGRAM TWOTHREE;
TYPE PTR =^NODE;
    NODE=RECORD
         LEFT,RIGHT,MIDDLE,PARENT:PTR;
         L,M,COUNT:INTEGER;
         END;
VAR N,F,V,TEMP,ROOT,K,FATHER,J,PRTY:PTR;
    NBR,A,NUM,MAX,NEWMAX:INTEGER;
    SEED:REAL;
    PRT:TEXT;
    COMD:CHAR;
    H:BOOLEAN;

    PROCEDURE UPDATE(VAR BOUND:PTR);
    BEGIN
     WHILE BOUND^.RIGHT^.COUNT = 0 DO  BOUND:=BOUND^.RIGHT;
    END;

    PROCEDURE ADDSON(Z:PTR);
    VAR X,X1,Z1:PTR;
    BEGIN(*CREATE NEW VERTEX AND MAKE RIGHTMOST TWO SONS OF 'Z'*)
     NEW(X);        (*LEFT AND RIGHT SONS OF 'X'  *)
     X^.LEFT:=TEMP;
     X^.RIGHT:=Z^.RIGHT;
     X^.MIDDLE:=NIL;
     X^.PARENT:=NIL;
     X^.COUNT:=0;
     TEMP^.PARENT:=X;
     Z^.RIGHT^.PARENT:=X;
     Z^.RIGHT:=Z^.MIDDLE;
     Z^.RIGHT^.PARENT:=Z;
     Z^.MIDDLE:=NIL;
     IF Z^.LEFT^.COUNT <> 0 THEN(*'Z' IS THE FATHER OF LEAF NODES*)
      BEGIN
      Z^.M:=Z^.RIGHT^.M;(* ADJUST L AND M VALUES OF Z  *)
      Z^.L:=Z^.LEFT^.M;
      END
    ELSE    (* 'Z' IS NOT THE FATHER OF LEAF NODES  *)
      BEGIN
      Z1:=Z^.LEFT;
      UPDATE(Z1);
      Z^.L:=Z1^.RIGHT^.M;(*ADJUST L AND M VALUES OF Z THRU LEAF*)
      Z1:=Z^.RIGHT;
      UPDATE(Z1);
      Z^.M:=Z1^.RIGHT^.M;
      END;
      IF X^.LEFT^.COUNT <> 0 THEN(*'X'IS THE FATHER OF LEAF NODES*)
         BEGIN
         X^.M:=X^.RIGHT^.M; (* ADJUST L AND M VALUES OF 'X' *)
         X^.L:=X^.LEFT^.M;
         END
```

```
ELSE     (* 'X'IS NOT FATHER OF LEAF NODES *)
   BEGIN
   X1:=X^.LEFT;
   UPDATE(X1);
   X^.L:=X1^.RIGHT^.M; (*ADJUST L AND M VALUES OF 'X' *)
   X1:=X^.RIGHT;
   UPDATE(X1);
   X^.M:=X1^.RIGHT^.M;
   END;
IF Z^.PARENT=NIL THEN (*'Z' IS ROOT.CREATE NEW ROOT AND*)
   BEGIN(*MAKE 'Z' LEFT SON OF ROOT,'X' RIGHT SON OF ROOT*)
     NEW(V);
     V^.LEFT:=Z;
     V^.RIGHT:=X;
     V^.MIDDLE:=NIL;
     V^.PARENT:=NIL;
     Z^.PARENT:=V;
     X^.PARENT:=V;
     V^.COUNT:=2;
     V^.L:=Z^.M;
     V^.M:=X^.M;
     ROOT:=V;
     V:=NIL;
   END
 ELSE                    (* 'Z' IS NOT ROOT *)
   BEGIN
     F:=F^.PARENT;

  IF F^.MIDDLE = NIL THEN (*FATHER OF 'Z' HAS TWO SON*)
    BEGIN
     X^.PARENT:=F;
     IF F^.LEFT=Z THEN
       BEGIN(*NEW VERTEX BECOMES MIDDLE SON OF FATHER*)
       F^.MIDDLE:=X;
       F^.L:=F^.LEFT^.M;       X:=NIL;
       F^.M:=F^.MIDDLE^.M;
       END
     ELSE  (* NEW VERTEX BECOMES RIGHT SON OF FATHER *)
       BEGIN
       F^.MIDDLE:=F^.RIGHT;
       F^.RIGHT:=X;
       F^.M:=F^.MIDDLE^.M;  X:=NIL;
       IF H THEN(*HIGHEST PRIORITY ITEM CAME INTO QUEUE*)
       BEGIN
        WHILE F^.PARENT <> NIL  DO
        BEGIN
        F:=F^.PARENT;
        IF (F^.MIDDLE=NIL) AND (F^.M < N^.COUNT) THEN
        F^.M:=N^.COUNT;
        END;
       END;
       END;
     END
   ELSE    (* FATHER OF 'Z' HAS THREE SONS *)
```

```
      IF F^.LEFT = Z THEN(*NEW VERTEX BECOMES SECOND SON *)
       BEGIN                  (* OF FATHER FROM LEFT. *)
       TEMP:=F^.MIDDLE;
       F^.MIDDLE:=X;    X:=NIL;
       ADDSON(F);
       END
      ELSE
        IF F^.RIGHT = Z THEN(*NEW VERTEX BECOMES FOURTH SON*)
        BEGIN                  (* OF FATHER FROM LEFT. *)
        TEMP:=F^.RIGHT;
        F^.RIGHT:=X;  X:=NIL;
        ADDSON(F);
        END
        ELSE(*NEW VERTEX BECOMES RIGHT SON OF FATHER *)
        BEGIN
        TEMP:=X;   X:=NIL;
        ADDSON(F);
        END;
END;
END;   (* END OF PROCEDURE ADDSON *)

FUNCTION SEARCH( A:INTEGER; R:PTR):PTR;
BEGIN
 IF R^.LEFT^.COUNT <> 0 THEN  (*RETURN POINTER TO VERTEX*)
      SEARCH:=R
 ELSE
  IF  A <= R^.L THEN  (*SEARCH THRU LEFT SON*)
      SEARCH:=SEARCH(A,R^.LEFT)
  ELSE
     IF (A<=R^.M) AND (R^.MIDDLE<>NIL) THEN
      SEARCH:=SEARCH(A,R^.MIDDLE) (*SEARCH THRU MIDDLE SON *)
     ELSE
      SEARCH:=SEARCH(A,R^.RIGHT); (* SEARCH THRU RIGHT SON *)
END; (*END OF FUNCTION SEARCH *)

PROCEDURE PRINT(TEST:PTR);
BEGIN
 IF TEST <> NIL THEN
 BEGIN
 IF  TEST^.COUNT <> 0 THEN
 BEGIN
 WRITE(PRT,'   ');
 WRITE(PRT,TEST^.M,'   ');
 END;
 PRINT(TEST^.LEFT);
 PRINT(TEST^.MIDDLE);
 PRINT(TEST^.RIGHT);
 END;
END;

FUNCTION RANDOM:INTEGER;(*GENERATES INTEGER  NUMBERS *)
BEGIN
 SEED:=SEED * 27.182813 + 31.415917;
 SEED:=SEED - TRUNC(SEED);
 RANDOM:=1 + TRUNC (80 * SEED);
END;
```

```
PROCEDURE INSERT(NUM:INTEGER);
BEGIN
  IF NBR=1 THEN(* THIS IS THE FIRST ITEM IN QUEUE *)
     BEGIN
     ROOT:=NIL;
     NEW(N);  (* CREATE NEW NODE  AND INITIALIZE  *)
     WITH N^  DO
      BEGIN
      COUNT:=NUM;
      L:=0;
      M:=COUNT;      PARENT:=NIL;
      LEFT:=NIL;
      RIGHT:=NIL;
      MIDDLE:=NIL;
      END;
      WRITELN(PRT,'RANDOM:',N^.COUNT);

     NEW(V); (* CREATE FIRST ROOT IN THE QUEUE AND MAKE *)
     WITH V^ DO  (* THE FIRST ITEM AS LEFT SON OF ROOT *)
     BEGIN
     COUNT:=0;
     LEFT:=N;
     RIGHT:=NIL;
     MIDDLE:=NIL;
     PARENT:=NIL;
     M:=0;
     END;
     V^.L:=N^.M;
     N^.PARENT:=V;
     ROOT:=V;
  END

ELSE
  IF  NBR = 2 THEN  (* SECOND ITEM CAME INTO QUEUE *)
     BEGIN
       NEW(N);
       WITH N^ DO
       BEGIN
       COUNT:=NUM;
       L:=0;
       M:=COUNT;
       LEFT:=NIL;    PARENT:=NIL;
       RIGHT:=NIL;
       MIDDLE:=NIL;
       END;
        WRITELN(PRT,'NBR: ',NBR,'  RANDOM: ',N^.COUNT);
       N^.PARENT:=ROOT;
       IF N^.COUNT>ROOT^.LEFT^.M  THEN
       BEGIN  (* MAKE 2th ITEM,RIGHT SON OF ROOT*)
        ROOT^.RIGHT:=N;
        ROOT^.M:=ROOT^.RIGHT^.M;
        END
```

```
      ELSE (*MAKE SECOND ITEM, LEFT SON OF ROOT *)
       BEGIN
       ROOT^.RIGHT:=ROOT^.LEFT;
       ROOT^.LEFT:=N;
       ROOT^.M:=ROOT^.L;
       ROOT^.L:=ROOT^.LEFT^.M;
       END;
    END
  ELSE   (* QUEUE HAS ALREADY 2 OR MORE ITEM IN IT *)
     BEGIN  (* CREATE NEW NODE AND INITIALIZE *)
     NEW(N);
     WITH N^ DO
     BEGIN
     COUNT:=NUM;
     L:=0;
     M:=COUNT;
     LEFT:=NIL;  PARENT:=NIL;
     RIGHT:=NIL;
     MIDDLE:=NIL;
     END;
     WRITELN(PRT,'NBR: ',NBR,'   RANDOM: ',N^.COUNT);

   F:=SEARCH(N^.COUNT,ROOT); (* POINTER TO THE FATHER OF *)
                             (* PROPER PLACE FOR NEW ITEM. *)
   IF F^.MIDDLE = NIL THEN    (* F HAS TWO SONS *)

      IF N^.COUNT <= F^.L  THEN
         BEGIN  (* MAKE THE NEW ITEM BE LEFT SON OF F *)
         F^.MIDDLE:=F^.LEFT;
         F^.LEFT:=N;
         N^.PARENT:=F;
         F^.M:=F^.MIDDLE^.M;
         F^.L:=F^.LEFT^.M;
         END
      ELSE
        IF N^.COUNT >= F^.M THEN
            BEGIN  (*MAKE NEW ITEM RIGHT SON OF F*)
            F^.MIDDLE:=F^.RIGHT;
            F^.RIGHT:=N;
            N^.PARENT:=F;
            IF (ROOT^.MIDDLE = NIL) AND (F = ROOT) THEN
              ROOT^.M:=N^.COUNT
            ELSE
              BEGIN
              WHILE F^.PARENT <> NIL DO
               BEGIN
               IF  F^.PARENT^.MIDDLE = NIL THEN
               F^.PARENT^.M:=N^.COUNT;
               F:=F^.PARENT;
               END;
              END;
         END
```

```
       ELSE  (*MAKE THE NEW ITEM MIDDLE SON OF F*)
          BEGIN
          F^.MIDDLE:=N;
          N^.PARENT:=F;
          F^.M:=F^.MIDDLE^.M;
          END
   ELSE       (* F ALREADY HAS THREE SONS *)
     IF N^.COUNT <= F^.L  THEN  (* MAKE NEW VERTEX *)
        BEGIN                 (*SECOND SON OF F *)
        TEMP:=F^.MIDDLE;
        F^.MIDDLE:=F^.LEFT;
        F^.LEFT:=N;
        N^.PARENT:=F;
        ADDSON(F);
        END
     ELSE
        IF N^.COUNT <= F^.M  THEN(* MAKE NEW VERTEX *)
                          (* SECOND SON OF F FROM LEFT*)
           BEGIN
           TEMP:=F^.MIDDLE;
           F^.MIDDLE:=N;
           N^.PARENT:=F;
           ADDSON(F);
           END
        ELSE
           IF N^.COUNT >  F^.RIGHT^.COUNT THEN
             BEGIN (*MAKE NEW VERTEX BE 4th SON OF F*)
              TEMP:=F^.RIGHT;
              F^.RIGHT:=N;
              H:=TRUE;
              ADDSON(F);
              H:=FALSE;
             END
           ELSE (*MAKE NEW VERTEX RIGHT SON OF 'F'*)
              BEGIN
              TEMP:=N;
              ADDSON(F);
              END;
        END;
    END;
```

```
PROCEDURE SUPSON;
VAR K1:PTR;
BEGIN
 FATHER:=K^.PARENT;
 IF  FATHER^.MIDDLE = NIL THEN (*FATHER HAS TWO SONS*)
   BEGIN
    J:=FATHER^.LEFT;
    IF J^.MIDDLE <> NIL THEN (*LEFT BROTHER HAS 3 SONS*)
      BEGIN
       K^.RIGHT:=K^.LEFT;
       K^.LEFT:=J^.RIGHT;
       K^.LEFT^.PARENT:=K;
       J^.RIGHT:=J^.MIDDLE;
       J^.MIDDLE:=NIL;

       IF K^.LEFT^.COUNT <> 0 THEN (*ADJUST L AND M VALUES*)
        BEGIN
        K^.M:=K^.RIGHT^.M;
        K^.L:=K^.LEFT^.M;
        END
       ELSE
        BEGIN
        K1:=K^.LEFT;
        UPDATE(K1);
        K^.L:=K1^.RIGHT^.M;
        K1:=K^.RIGHT;
        UPDATE(K1);
        K^.M:=K1^.RIGHT^.M;
        END;
        FATHER^.L:=J^.M;
        FATHER^.M:=K^.M;
        WHILE FATHER^.PARENT <> NIL DO
         BEGIN
         FATHER:=FATHER^.PARENT;
         IF FATHER^.MIDDLE = NIL  THEN  FATHER^.M:=K^.M;
         END;
     END
    ELSE         (* LEFT BROTHER HAS TWO SONS *)
     BEGIN
       J^.MIDDLE:=J^.RIGHT;
       J^.RIGHT:=K^.LEFT;
       J^.RIGHT^.PARENT:=J;
       FATHER^.RIGHT:=NIL;
       K:=FATHER;
       IF K = ROOT THEN   (* WE REACHED THE ROOT *)
        BEGIN
        ROOT:=ROOT^.LEFT;
        ROOT^.PARENT:=NIL;
        END
      ELSE SUBSON;(*NOT REACHED TO ROOT,FATHER HAS ONE SON*)
     END;
END
```

```
ELSE   (*FATHER HAS 3 SON *)
  BEGIN
  J:=FATHER^.MIDDLE;

  IF J^.MIDDLE=NIL THEN (*MIDDLE BROTHER HAS TWO SONS*)
    BEGIN
     J^.MIDDLE:=J^.RIGHT;
     J^.RIGHT:=K^.LEFT;
     J^.RIGHT^.PARENT:=J;
     FATHER^.RIGHT:=FATHER^.MIDDLE;
     FATHER^.MIDDLE:=NIL;
     FATHER^.M:=K^.L;
     WHILE FATHER^.PARENT <> NIL DO
     BEGIN
      FATHER:=FATHER^.PARENT;
      IF FATHER^.MIDDLE = NIL THEN  FATHER^.M:=K^.L;
     END;
    END

   ELSE  (* MIDDLE BROTHER HAS THREE SONS *)
     BEGIN
     K^.RIGHT:=K^.LEFT;
     K^.LEFT:=J^.RIGHT;
     K^.LEFT^.PARENT:=K;
     J^.RIGHT:=J^.MIDDLE;
     J^.MIDDLE:=NIL;
     K^.L:=K^.LEFT^.M;
     K^.M:=K^.RIGHT^.M;
     WHILE FATHER^.PARENT <> NIL DO
      BEGIN
      FATHER:=FATHER^.PARENT;
      IF FATHER^.MIDDLE = NIL  THEN  FATHER^.M:=K^.M:
      END;
     END;

 END;
END;
```

```
PROCEDURE DELETE; (*REMOVES THE RIGHT MOST NODE*)
BEGIN
K:=ROOT;
IF(K^.LEFT=NIL) AND (K^.RIGHT=NIL) THEN WRITELN(PRT,'NO ITEM')
ELSE
  IF (K^.MIDDLE=NIL) AND (K^.RIGHT^.COUNT <> 0) THEN
    BEGIN  (* THERE ARE ONLY TWO ITEMS IN THE QUEUE *)
    MAX:=K^.RIGHT^.COUNT;
    K^.RIGHT:=NIL;
    WRITELN(PRT,'MAX: ',MAX);
    IF K^.LEFT=NIL THEN(*THERE IS ONLY ONE ITEM IN THE QUEUE*)
    WRITELN(PRT,'LAST ITEM')
    ELSE
     BEGIN
     K^.RIGHT:=K^.LEFT;
     K^.LEFT:=NIL;
     END;
    END
  ELSE  (* THERE ARE MORE THAN TWO ITEMS IN THE QUEUE *)
      BEGIN
      UPDATE(K);
      MAX:=K^.RIGHT^.COUNT;
      WRITELN(PRT,'MAX:   ',MAX);
      IF K^.MIDDLE = NIL THEN  SUBSON (* K  HAS TWO SONS *)
      ELSE                    (* K  HAS THREE SONS *)
        BEGIN
         K^.RIGHT:=NIL;
         K^.RIGHT:=K^.MIDDLE;
         K^.MIDDLE:=NIL;
         NEWMAX:=K^.RIGHT^.COUNT;
         WHILE K^.PARENT <> NIL DO
          BEGIN
          K:=K^.PARENT;
          IF K^.MIDDLE = NIL THEN      K^.M:=NEWMAX;
          END;
         END;
       END;
    END;


    PROCEDURE BEST;(*RETURNS RIGHT MOST ITEM IN THE QUEUE*)
    VAR PRTY:PTR;
        BIG:INTEGER;
    BEGIN
     PRTY:=ROOT;
     IF PRTY^.RIGHT <> NIL THEN
     BEGIN
      UPDATE(PRTY);
      BIG:=PRTY^.RIGHT^.COUNT;
      WRITELN(PRT,'HIGHEST PRIORITY IN QUEUE IS :   ',BIG);
     END
     ELSE WRITELN(PRT,'QUEUE IS EMPTY..');
    END;
```

```
BEGIN(*MAIN*)
REWRITE(PRT,'CONSOLE:');
SEED:=0.20002;
NBR:=0;
E:=FALSE;
WHILE NBR <250 DO
BEGIN
READ(COMD);
IF COMD = 'I' THEN      (* INSERT COMMAND *)
 BEGIN
 NBR:=NBR+1;
 NUM:=RANDOM;
 INSERT(NUM);
 END
ELSE
  IF COMD = 'D'  THEN   (* DELETE COMMAND *)
   BEGIN
    NBR:=NBR-1;
    DELETE;
   END
   ELSE
    IF COMD = 'B' THEN  BEST (* FIND HIGHEST PRIORTY*)
    ELSE WRITELN(PRT,'INVALID COMMAND');
 END;
END.
```

H. FIXED PRIORITY

```
(* THIS IS THE IMPLEMENTATION OF A PRIORITY QUEUE BY *)
(* USING A FIX PRIORITY PROPERTY. A DATA TYPE RECORD *)
(* IS USED TO REPRESENT THE NODES. THE ID FIELD IN   *)
(* THE RECORD INDICATES THE IDENTIFICATION OF A ITEM.*)

PROGRAM FIXPRTY;
CONST MAX=50;
      N=13;      (*NUMBER OF DIFFERENT PRIORITIES*)
      M=25;
TYPE PTR=^CIT;
     CIT=RECORD
         ID:INTEGER;
         NEXT:^CIT;
         END;
    NODE=RECORD
         FIRST,LAST:^CIT;
         END;
     A=SET OF 1..M;
VAR B:ARRAY[13..M] OF NODE;  (*INDEX OF EXTERNAL NODES*)
    TOTAL:A;
    HEIGTE,X,MAXIM,NUM,PRTY,K:INTEGER;
    COMD:CHAR;
    SEED:REAL;
    Y,Z:PTR;


PROCEDURE INSERT(K:INTEGER); (* INSERTS THE NEW NODE *)
BEGIN
  NUM:=NUM+1;
  NEW(Y);        (*CREATE NEW NODE FOR NEW ITEM*)
  Y^.ID:=22;     (*NEW ITEM IDENTIFICATION*)
  Y^.NEXT:=NIL;
  IF B[K].LAST= NIL   THEN
  BEGIN          (*FIRST ITEM IN THIS PRIORTY*)
   B[K].FIRST:=Y;  (*LINK NEW ITEM*)
   B[K].LAST:=Y;
   REPEAT        (*SET UP ARRAY ALONG PATH THRU ROOT*)
     TOTAL:=TOTAL + [K];
    K:=K DIV 2;
   UNTIL K=0;
  END
  ELSE           (*THERE IS ATLEAST ONE ITEM IN THIS PRIORTY*)
   BEGIN
     B[K].LAST^.NEXT:=Y;  (*LINK NEW ITEM AS LAST ITEM*)
     B[K].LAST:=Y;
    END;
END;
```

```
PROCEDURE DELETE; (* REMOVES THE NODE WITH HIGEEST PRTY.*)
VAR J:INTEGER;
BEGIN
  IF NOT (~1 IN TOTAL )   THEN WRITELN('NO ITEM IN QUEUE')
  ELSE
    BEGIN
     NUM:=NUM -1;
     J:=1;
     WHILE  J < N DO
     BEGIN  (* FIND THE HIGEEST PRIORITY IN QUEUE*)
      J:=2*J;
      IF J+1 IN TOTAL  THEN  J:=J+1; (*GO THRU RIGHT SON*)
     END;
     IF B[J].FIRST <> B[J].LAST THEN (*THERE ARE AT LEAST 2*)
       B[J].FIRST:=B[J].FIRST^.NEXT (*ITEM IN THIS PRIORTY*)
     ELSE
       BEGIN    (*THERE IS ONLY ONE ITEM IN THIS PRIORITY*)
        B[J].FIRST:=NIL;
        B[J].LAST:=NIL;
        TOTAL:=TOTAL - [J];
        REPEAT
          IF (J MOD 2) = 0 THEN
           BEGIN (*WE ARE ON THE LEFT SON,SINCE RIGHT SON IS*)
            J:=J DIV 2;        (* ZERO, SET ITS ROOT ZERO*)
            TOTAL:=TOTAL-[J];
           END
          ELSE
           IF J-1 IN TOTAL THEN J:=1(*WE ARE ON THE RIGHT SON*)
           ELSE                        (*DON'T CHANGE ITS ROOT*)
            BEGIN      (*LEFT SON IS ZERO,SET ROOT ZERO*)
             J:=J DIV 2;
             TOTAL:=TOTAL-[J];
            END;
        UNTIL J=1;  (* WE REACHED ROOT,TERMINATE..*)
       END;
    END;
  END;

  FUNCTION RANDOM:INTEGER; (*GENERATES RANDOM NUMBERS*)
  BEGIN
  SEED:=SEED * 27.182813 + 31.415917;
  SEED:=SEED - TRUNC(SEED);
  RANDOM:=1 + TRUNC(N * SEED);
  END;


  PROCEDURE PRINT;
  VAR V:INTEGER;
  BEGIN
   FOR V:=1 TO M DO
    BEGIN
    IF V IN TOTAL THEN  WRITE(V,' ')
    ELSE WRITE('=');
    END;  WRITELN;
  END;
```

```
BEGIN (*MAIN*)
SEED:=0.2;
NUM:=0;
TOTAL:=[];
FOR X:=11 TO M DO
BEGIN   (* INITIALIZE EXTERNAL NODES *)
 B[X].FIRST:=NIL;
 B[X].LAST:=NIL;
END;
HEIGTH:=0;
X:=N;
REPEAT   (* FIND (HEIGHT-1) OF TREE*)
 X:=X DIV 2;
 HEIGTH:=HEIGTH + 1;
UNTIL X=1;
MAXIM:=2;
REPEAT  (*FIND RIGHT MOST POSITION ON (HEIGHT-1)*)
MAXIM:=2 * MAXIM;
HEIGTH:=HEIGTH - 1;
UNTIL HEIGTH=0;
MAXIM:=MAXIM - 1;
WHILE NUM < MAX DO
 BEGIN
  WRITE('>');
  READLN(COMD);
  IF COMD = 'I' THEN
   BEGIN
   PRTY:=RANDOM;  WRITELN('RANDOM:',PRTY);
   X:=MAXIM + PRTY;(*PROPER INDEX FOR THE NEW ITEM*)
   IF X > M THEN
    BEGIN
    X:=(X-M) + (N-1);
    INSERT(X);
    END
   ELSE INSERT(X);
   END
  ELSE
   IF COMD = 'D' THEN  DELETE
   ELSE  WRITELN('INVALID COMMAND');
   PRINT;
 END;
END.
```

LIST OF REFERENCES

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass. (1975).

2. A. Kerschenbaum and R. Van Slyke, "Computing Minumum Spanning Trees Efficiently," Proc. 25th Annual Conference of the ACM, 1972, 518-527.

3. B. L. Fox, "Accelerating List Processing in Discrete Programming," J.ACM 17, 2(april 1970), 383-384.

4. Brown Mark Robbin, The Analysis of a Practical and Nearly Optimal Priority Queue, Ph. D. Thesis, Standford University, 1977.

5. Donald B. Johnson, "Efficient Algorithms for Shortest Paths in Sparce Networks," J.ACM 24, 1(Janaury 1977), 1-13.

6. Ellis L. Johnson, "On Shortest Paths and Sorting," Proc. 25th Annual Conference of The ACM, 1972, 510-517.

7. Donald E. Knuth, The Art of Computer Programming, Vol.1, Fundamental Algorithms, Addison-Wesley, Reading, Mass. (1973).

8. Donald E. Knuth, The Art of Computer Programming, Vol.3, Sorting and Searching, Addison-Wesley, Reading, Mass. (1975).

9. G. M. Adel'son, Vel'skii and E. M. Landis, "An Algorithm for the organization of information", Dokl. Akad. Nauk SSSR 146, 263-266.(in Rusian). English Translation in Soviet Math. Dokl. 3(1962), 1259-1262.

10. Peter C. Brillinger, Doran J. Cohen, Introduction to Data Structures and Non-Numeric Computation, Prentice-Hall, Inc. (1977).

11. Nikaus Wirth, Algorithms+Data Structures=Programs, Prentice-Hall, Inc. (1976).

12. Sara Baase, Computer Algorithms:Introduction to design and Analysis, Addison-Wesley Series in Computer Science, (1978).

13. Thomas A. Standish, Data Structure Techniques, Addison-Wesley series in Computer science.(1980).

14. Ellis Horowitz, Sartaj Sahni, Fundamentals of Data Structures, Computer Science Press, Inc. (1976).

15. Jean Francon, Gerard Viennot, Jean Vuillemin, "Description et Analyse D'une Representation Performante Des Files de Priorite," Laboratoire de recherche en Informatique et Automatique, Rapport de Recherche N0.287, Avril.(1978).

16. B. L. Fox, J. K. Lenstra, A. H. G. Rinnooy Kan, L. E. Schrage, "Branching from the Largest Upper Bound", European Journal of Operational Research 2(1978), 191-194.

INITIAL DISTRIBUTION LIST

		No. copies
1.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
2.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
3.	Ass. Professor D. R. Smith, Code 52Sc Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Turk Deniz Kuvvetleri Komutanligi Egitim Dairesi Baskanligi ANKARA - TURKEY	2
5.	Golcuk Otomatic Bilgi Islem Merkezi GOLCUK - KOCAELI - TURKEY	1
6.	Bogazici Universitesi Komputer Bolumu BEBEK - ISTANBUL - TURKEY	1
7.	Ortadogu Teknik Universitesi Komputer Bolumu ANKARA - TURKEY	1
8.	Dz. Utgm. Alinur Goksel Ahmet Vefik Pasa cad. Topyay Apt. No: 72/4 CAPA - ISTANBUL - TURKEY	2
9.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2